Biological Database Indexing and its Applications

Cheung Ching Fung

August 30, 2002

香港中文大學
THE CHINESE UNIVERSITY OF HONG KONG

A Thesis Submitted in Partial Fulfilment
of the Requirements for the Degree of
Master of Philosophy
in
Systems Engineering and Engineering Management


香港中文大學
圖書館系統
07 MAR 2003
THE UNIVERSITY OF HONG KONG

摘要

生物學的資料庫索引及式樣發掘是資料庫和知識發掘的新趨勢。在不同的生物學實驗中所取得的大量脫氧核糖核酸和蛋白質數列，需要有效率的資料庫索引和式樣發掘，才能把數據轉化爲知識，在生物學和醫學上爲人類帶來突破，加深我們對人體和疾病的了解。

後置樹是一種在數據處理中常用的數據結構，它把一個字串中的所有 後置詞也索引在一起。這篇論文提出了一個新穎的後置樹建立方案，解決了現有的效率問題。我們指出了在現有的後置樹建立中所出現的 **邊線分割**，**隨機存取** 和 **數據歪斜**等問題，也提出 了一個可用極少內存記憶體來實現極大的索引的方法。它不但大大減 低對系統上的需求，同時也大大提高了對查詢和應用上的效率。

我們的實驗結果證明我們足以在三個小時內以16MB的內存記憶體來實 現多達230MB的脫氧核糖核酸數據索引，以這樣小的內存記憶體，在目前的所有方案都不可能實現。

Abstract

Biological database indexing and pattern discovery is a new trend in database and knowledge discovery in database. Given the huge amount of DNA and proteins sequences gathered from various experiments, efficient indexing and pattern discovery is of great urgency and importance.

Suffix trees have been frequently used as the database index for biological sequences and this thesis proposed a new suffix tree construction as a remedy for the current inefficiency. We address the problems of *Edge Splitting*, *Randem Access*, and *Data Skew* in the current persistent suffix tree construction approach, and we propose a new approach to create a huge suffix tree using a very small amount of main memory and provide a superior performance in string matching queries and applications.

Our experimental results show that our approach can utilize a 16MB main memory to index a 230 Mbps DNA sequence in 3 hours while the best existing method cannot ever complete in such a small amount of main memory.

Acknowledgements

I would like to take this opportunity to thank my supervisor, Jeffrey, Xu Yu, for his patience and valuable advice in my research. Although the alacity from excellence in my research is memorable, the process is arduous and harsh. Fortunately, Professor Yu supports and encourages me to pass through this arduous road. In addition to his patience and advice, he also taught me how to analyse a problem, and how to present a work. A work without good presentation is worthless.

I would like to thank Professor Hongjun Lu. Professor Lu is from Department of Computing Science, the Hong Kong University of Science and Technology. He gave me many valuable advice and ideas. I would also like to take this opportunity to thank Doctor Ela Hunt. She is from the Department of Computing, the University of Glasgow. She is very helpful and gave us a lot of advice and feedbacks.

I would like to thank my thesis committee, Professor Wai Lam, Christopher Chuen-Chi Yang from the Department of Systems Engineering and Engineering Management, for their precious comments and feedback.

I wish to thank my parents, who have given me concerns and support everytime. They provided a warm family and a comfortable environment for me.

Finally, I would also like to thank everyone in the Department of Systems Engineering and Engineering Mangement. Fiona, Gabriel, Holmes, Paul and Silvia are all my wonderful colleagues. Without their support, my study in Master degree would be boring and lonely.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Ever since the structure of DNA was unraveled in 1953, molecular biology has witnessed tremendous advances. Recent advances in sequencing technology have allowed the rapid accumulation of DNA and protein data. A huge amount of bio-sequences has been and is being generated in laboratories all over the world. The entire DNA of an organism comprises that organism's genome. The human genome has an estimated 3×10^9 base pairs. Effective data structures and efficient algorithms are highly required to support exact/approximate sequence matching as well as to find repetitive structures in DNA sequences. Suffix tree is such a data structure which can be constructed in linear time $O(n)$, if it can fit in main memory [11, 12, 20]. Many efficient in-memory suffix-tree construction algorithms were studied [6, 7, 8, 9, 10, 19, 17, 16].

It is widely acknowledged that suffix-trees are difficult to be built on disk. Previous efforts on constructing disk-based suffix-trees included Bieganski [3]. Baeza-Yates and Navarro constructed disk-based suffix-trees for sequences of 1Mbps using a machine with small memory (64MB) [2, 1] and concluded that it is infeasible to construct a suffix-tree that exceeds the main memory available. In [8], Farach and Muthukrishnan pointed out that random access to the string being indexed is the main bottleneck.

In [14], Hunt et al proposed a pre-partitioning disk-based suffix-tree con-

struction algorithm which abandons the use of suffix-links and pre-partition the sequence into m-subsequences, based on the main-memory available. We call it a `PrePar-Suffix` algorithm in this thesis, because it uses a pre-partitioning technique. `PrePar-Suffix` performs m passes over the original sequence, and enlarges the suffix-tree for each of the m subsequences each time. They successfully indexed a 286 Mbps DNA sequence using 2GB main memory. However, it cannot handle data skew because it partitions a DNA sequence into even partitions under the assumption of pseudo-random nature of DNA sequences. It exhibits exponential I/O cost when data skew occurs.

1.1 Biological Sequences

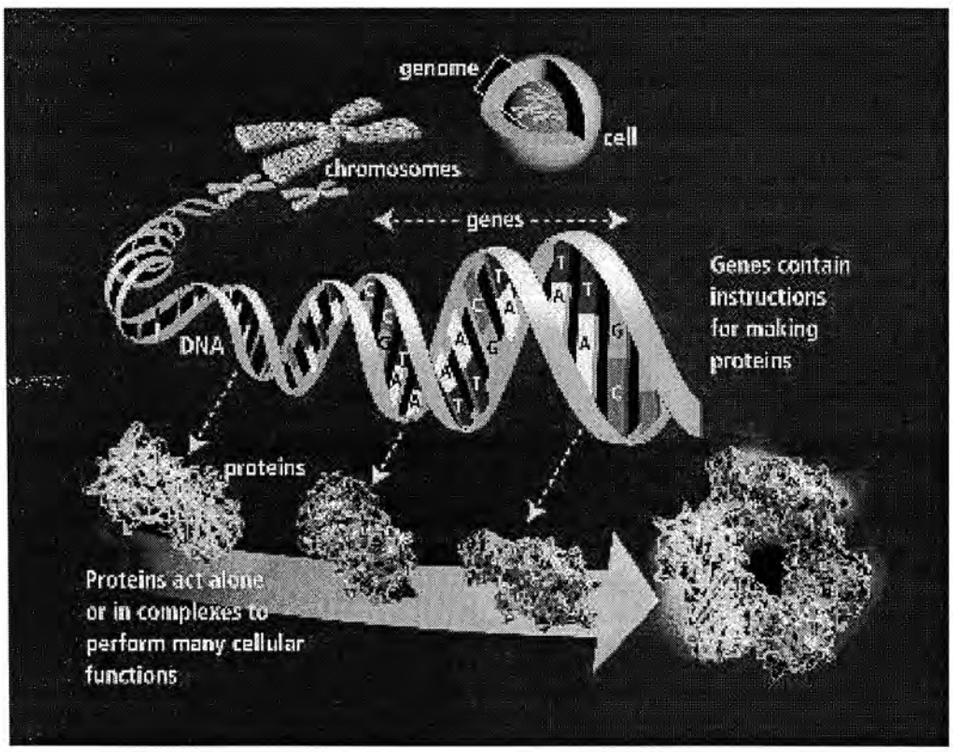


Figure 1.1: DNA Sequence and Protein (Source: US Department of Energy)

Genome is the complete set of instructions for making organisms and it is the blueprint for all cellular structures and biological activities. Cells are

the fundamental of working units of all living organisms and their activies are directed by the instructions from genome. Figure 1.1 illustrates the relationship between genome and living organisms. Inside the nucleus of a cell, genome consists of tightly coiled threads of *deoxyribonucleic acid* (DNA) and the associated protein molecules. Genome is organized into structures that are called *chromosomes.*

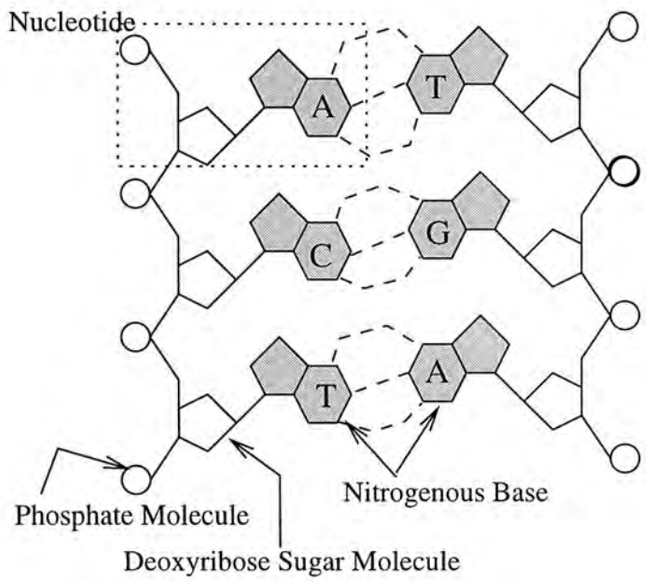


Figure 1.2: DNA Structure

Apart from reproductive cells and mature red blood cells, every cell in the human body contains 24 distinct pairs of chromosomes. In human and other higher organisms, a chromosome consists of two strands that wrapped around each other. Figure 1.2 shows the structure of a DNA molecule. The strands are made up of sugar and phosphate molecules, which are called the *sugar-phosphate backbone*, and the two strands are connected by nitrogen containing chemicals called as *bases.* Four different nitrogenous bases are present in DNA. They are adenine (A), thymine (T), cytosine (C), and guanine (G). One sugar, one phosphate, and one nitrogenous base together form a *nucleotide*, and the linear arrangement of nucleotides forms a strand of DNA molecule. The particular order of bases arranged along a sugar-

phosphate backbone is called a *DNA Sequence*. A DNA sequence carries the exact genetic instructions that are required to create a particular organism with its own characteristics.

A DNA molecule contains many *genes*, and a gene is the basic physical and functional unit of heredity. The base sequence inside a gene encoded the instruction for proteins synthesis. Proteins are large and complex molecules which are made up of 20 amino acids, and they are the structural and functional components cells that are responsible for many life activities. However, genes only comprise of 2% of the total human genome and the remainders consists of non-coding regions whose functions are obscure.

In October 1990, the Human Genome Project formally began and its aim is to discover all the estimated 30,000-35,000 genes and 3 billion bases in human genome for biological researches and studies. Finally, in December 2001, all the chromosomes in human genome were sequenced and our experimental sequences are obtained from the *National Center for Biotechnology Information* [1].

1.2 User Queries on Biological Sequences

There are three types of user queries on biological sequences. They are the *Exact Sequence Matching*, *Approximate Sequence Matching*, and the *Finding of Repetitive Structures*. Exact sequence matching concerns about finding all occurrences of a pattern in a sequence and it is the simplist type of user queries. For example, we would like to find all occurrences of the pattern AC in the sequence $ACAACC$. The pattern occurs in the positions 1 and 4 in the sequence. There are already a number of algorithms that can perform exact seqeunce matching in linear time. For example, the Boyer-Moore and Knuth-Morris-Pratt algorithms can find all occcurences in time proportional to the total length of sequence and pattern. With the help of a database index, the running time of such a query can be reduced to time proportional

[1] ftp://ncbi.nlm.nih.gov/genomes/H_sapiens

to the length of pattern.

Approximate sequence matching is more complicated than the exact sequence matching problem. When we allow for at most k mismatches (substitutions) in finding all occurrences of a pattern, the problem is called the k-mismatches approximate sequence matching. When we relax the constraint and allow for insertions and deletions, the problem becomes the k-difference approximate sequence matching. Approximate sequence matching is of great importance because mutations always occur in the biological sequences, and similar genes tend to exhibit similar physical and functional behavior. For example, when we would like to find the pattern ACA in the sequence $ACAACC$ and allow 1 mismatch, we would get positions 1 and 4. The problem of approxiate sequence matching can be answered using dynamic programming and edit distance. The database index can help to reduce the amount of computation and speed up the query time.

Biological sequences possess many repetitive structures. For instacne, tandem repeats and parlindromes are very common in DNA sequences.

Tandem repeats are closely related to DNA fingerprinting, genotyping and inherited diseases. Although tandem repeats do not participate in protein synthesis, they carry sufficient genetic information and are unique among individuals. Tandem repeats are classified by the length of their repeating units. Tandem repeat units with size between 2 to 5 bases are called microsatellites, 9 to 80 bases are called mini-satellites, and 100 to 300 are called satellites. When a repeated unit repeats more than twice, the tandem repeats are called as Variable Number of Tandem Repeats (VNTR). There exists a linear time algorithm in finding all tandem repeats occurrences in a biological sequence using a database index. Without the help of a database index, it would be difficult to perform the query efficiently.

1.3 Research Contributions

The significant contribution of this thesis is that we propose an algorithm for large database index cosntruction using a dynamic clustering technique. Our algorithm exhibits $O(n \log n)$ behavior in terms of CPU cost and linearity of I/O cost. We can construct disk-based suffix-trees for over 200Mbps DNA sequences using only 16MB memory. We completely solve the so-called memory bottleneck problems when constructing large disk-based suffix- trees for huge DNA sequences.

1.4 Organization of Thesis

The rest of the thesis is organized as follows. In Chapter 2 discusses the basic notion of suffix-tree and outlines our two-phase strategy and shows the disk-based suffix-tree and suffix-link index. In Chapter 3, we first introduce the existing disk-based suffix-tree construction algorithm `PrePar-Suffix` which uses a pre-partitioning technique. Then, we will identify the three issues with pre-partitioning techniques. We will discuss our novel `DynaCluster-Suffix` which uses a dynamic clustering technique. We also discuss how to rebuild suffix-links for the disk-based suffix-trees using least common answer queries (*lca*-queries) in Chapter 4, and how to preform exact sequence matching in Chapter 5. In Chapter 6, we will give a brief introduction to our system and outline our data structures and other implementation issues. We will report our extensive experimental results in Chapter 7. We conclude the thesis in Chapter 8.

Chapter 2

Background

In this chapter, we formally introduce the suffix tree as the data structure for biological database indexing. Suffix trees can support a large varieties of applications. For instance, suffix trees can be used in exact/approximate sequence matching and finding repetitive patterns in a sequence. Furthermore, when a suffix tree can be fitted in main memory, it can be built in linear time $O(n)$ efficiently. Following that, we will give a brief introduction to our two-phase suffix tree construction strategy, and show a disk-based suffix tree with the suffix-link index.

2.1 What is a Suffix-Tree?

In this thesis, we use the similar notions used in [6, 8]. Let Σ denote a set of alphabets, and $|\Sigma|$ denote the size of the set of alphabets. Let $S = s_1 s_2 \cdots s_n$ denote a sequence of length n, where $s_i \in \Sigma$. A suffix of a sequence S, s_i, is the subsequence stretching from s_i to the end of the sequence $[s_i \cdots s_n]$. A suffix number, i, is the position of s_i in the sequence. For DNA, $\Sigma = \{$`A`, `C`, `G`, `T`$\}$ and $|\Sigma| = 4$. A sample DNA sequence is `AACCAA`. Here, $s_1 =$ `AACCAA`, $s_2 =$ `ACCAA`, and $s_6 =$ `A`. A suffix tree, denoted T_S, for a sequence S of length n is a rooted directed tree with exactly n leaf nodes. A leaf node l_i represents a suffix s_i. Each non-leaf node, other than the root, has at least two children,

and each edge is labelled with a nonempty subsequence of S. The length of edge is defined as the length of edge label (the subsequence). No two edges out of a node can have the same leading alphabet in the edge labels. Each non-leaf node v in T_S has a length $L(v)$ which is the sum of the edge lengths on the path from the root to v. The sequence at node v, denoted $\sigma(v)$, is $[s_i, \cdots s_{i+L(v)-1}]$, for any leaf node l_i below v. We show a Lemma below.

Lemma 1 *([7, 23]) Let a be an alphabet and α be an arbitrary subsequence of S. If there is a node v in T_S such that $\sigma(v) = a\alpha$, then there is a node w in T_S such that $\sigma(w) = \alpha$.*

Based on Lemma 1, for every node v in a suffix tree, the suffix link, $sl(v) = w$, where v and w are defined as in Lemma 1. Let $lcp(\alpha, \beta)$ be the length of the longest common prefix of two sequences, α and β. Let $lca(v, w)$ be the least common ancestor of any two nodes v and w in a suffix tree T_S. The property of suffix trees often exploited algorithmically is the following relationship between lcp in S and lca in T_S: for all $v, w \in T_S$, $lcp(\sigma(v), \sigma(w)) = |\sigma(lca(v, w)|$.

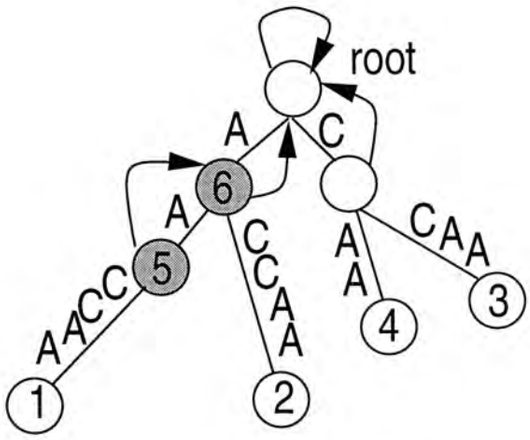


Figure 2.1: A suffix tree T_S for $S =$ `AACCAA`

2.2 Disk-Based Suffix-Trees

Figure 2.1 shows an image of in-memory suffix tree. For dealing with large DNA sequences over 200Mbps, disk-based suffix trees are required. In this thesis, we show a novel way to construct a disk-based suffix tree efficiently. A disk-based suffix tree for the same simple sequence AACCAA is shown in Figure 2.2, to be stored in a biological database. The right shows the sequence AACCAA with its suffix number, i, on the top of each alphabet. In the middle is the suffix tree without suffix links. The number inside a node indicates the physical node identifier. The number i next to a node, v, indicates that the node, v, represents a suffix s_i. For example, the node, with a node identifier 3, represents a suffix s_1 = AACCAA. The suffix-links are handled as suffix-link indices shown on the left. The numbers shown in the suffix links are node-identifiers in the suffix tree in the middle. For example the node 2 in the suffix tree (in the middle) has a suffix link pointing to the node 1. We store suffix-links for the suffix tree separately. Therefore, there is no overhead for applications that do not need to use suffix links.

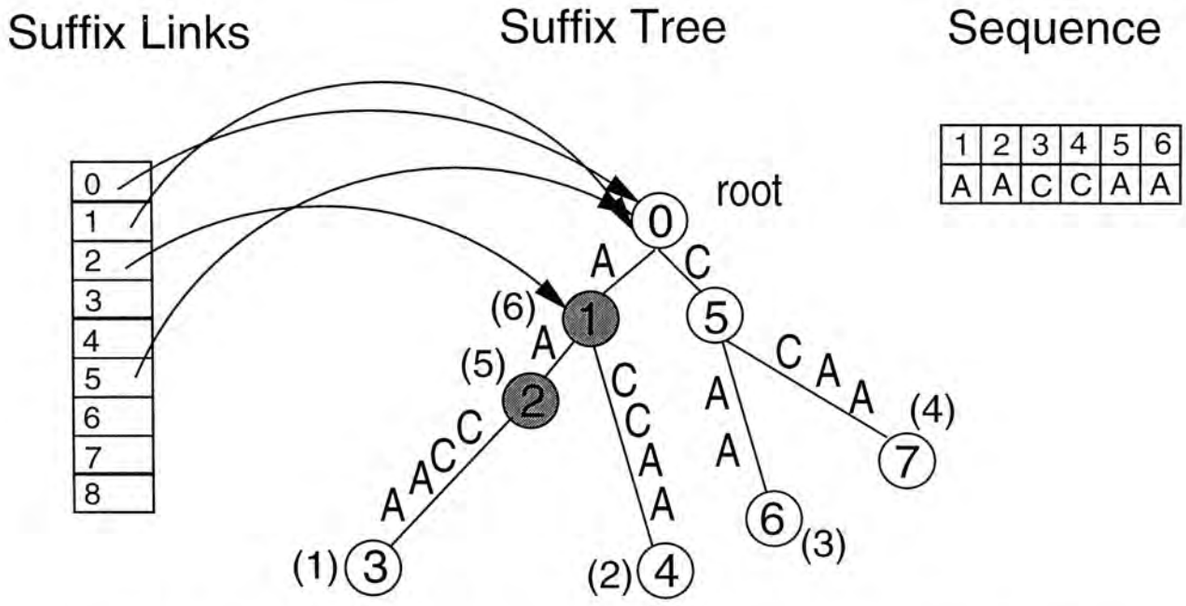


Figure 2.2: The disk-based suffix tree for AACCAA in a biological database

In the following, we discuss a two phase disk-based suffix tree construction strategy. In the first phase, we construct a disk-based suffix tree without suffix links, and in the second phase, we rebuild the suffix-links. We give some notions that will be used for disk-based suffix-tree construction. A function c is called the *character code* of an alphabet in the set alphabets (Σ). For DNA, $c(\texttt{A}) = 0$, $c(\texttt{C}) = 1$, $c(\texttt{G}) = 2$ and $c(\texttt{T}) = 3$. A prefix of a suffix s_k of length l is a subsequence: $s_i s_{i+1} \cdots s_{i+l-1}$, denoted s_k^l. We call s_k^l the prefix of suffix s_k of length l. For example, $s_1 = \texttt{AACCAA}$, then $s_1^3 = \texttt{AAC}$.

Chapter 3

Disk-Based Suffix Tree Constructions

In this chapter, we will first review the previous disk-based suffix tree construction, the `PrePar-Suffix` [14]. The `PrePar-Suffix` divides a suffix tree into a number of partitions. The number of partitions depends on the total memory instantiation for the whole DNA sequence and the total memory available. Hunt has used the `PrePar-Suffix` to construct a suffix tree for a 289Mbps DNA sequence. However, as we will discuss in this chapter, the `PrePar-Suffix` suffers from the problems of edge splitting, data skew, and random access. We will also introduce our new suffix tree construction, `DynaCluster-Suffix`, in the end of this chapter.

3.1 An Existing Algorithm: `PrePar-Suffix`

`PrePar-Suffix` does not use suffix links during the construction. `PrePar-Suffix` assumes the pseudo-random nature of DNA, and assumes that the tree is uniformly populated. It divides the DNA sequence into m-partitions. The number of partitions is $m = \lceil M_S/M_A \rceil$ where M_S and M_A are the total memory instantiation for the whole DNA sequence and the total memory available, respectively. The *prefix code* of a prefix of sufficient length l,

Prefix	Prefix Code	Prefix	Prefix Code
AA	0	GA	8
AC	1	GC	9
AG	2	GG	10
AT	3	GT	11
CA	4	TA	12
CC	5	TC	13
CG	6	TG	14
CT	7	TT	15

Table 3.1: Prefix codes for prefixes of length 2

$s_i^l = [s_i s_{i+1} \cdots s_{i+l-1}]$, is calculated by

$$P(s_i^l) = \sum_{j=0}^{l-1} c(s_{i+j}) \cdot |\Sigma|^{l-j-1} \tag{3.1}$$

The minimum value for $P(s_i^l)$ is 0 and its maximum value is $|\Sigma|^l - 1$. So the range of prefix code for each of the m-partition, r, is given by $r = \lceil (|\Sigma|^l - 1)/m \rceil$.[1] `PrePar-Suffix` scans the whole sequence m times. The suffixes that are indexed during the j-th scan of the sequence are in the range $j \cdot r \leq P(s_i^l) < (j+1) \cdot r$. Table 3.1 shows the prefix codes for prefix length 2. Suppose $l = 2$ and $m = 2$, then r becomes 8. That is, when `PrePar-Suffix` scans a DNA sequence, in the first scan, it will deal with the prefixes in the first column of Table 3.1, and, in the second, it will deal with the prefixes in the third column.

The `PrePar-Suffix` algorithm is shown in Algorithm 1. Consider the sample DNA sequence: `AACCAA`. Suppose it needs to be divided into $m = 4$ subsequences. Let the sufficient prefix length be $l = 1$. In the first scan, `PrePar-Suffix` handles four suffixes: `AACCAA`, `ACCAA`, `AA` and `A`, and, in the second scan, it handles two suffixes: `CCAA` and `CAA`, as illustrated in Figure 3.1. No nodes will be added into the suffix tree during the last two scans, because there are no `G`s and `T`s in the sample sequence. Initially, `Prepar-Suffix`

[1] When $r = 1$, given m-partitions of a sequence, a *prefix of sufficient length*, l, is then calculated as, $l = \lceil \log_a m \rceil$.

Algorithm 1 `PrePar-Suffix`

Input: a DNA sequence of length n: $s_1 s_2 \cdots s_n$.
Output: a suffix tree.

```
Suppose the sequence is divided into m-subsequences;
Determine sufficient prefix length l and r;
for j = 0 to m − 1 do
  if the prefix code of s_i^l, P(s_i^l), is in the range of j · r ≤ P(s_i^l) < (j + 1) · r
  then
    traverse from the root;
    insert s_i into the suffix tree at the mismatching position;
    do edge-splitting if needed;
  end if
end for
```

creates a root node for the suffix tree T_S numbered 0. Then it inserts the suffix s_1 = `AACCAA` into T_S. Traversing from the root, `PrePar-Suffix` finds out it needs to add a new node for s_1. Next, when inserting s_2 = `ACCAA`, `PrePar-Suffix` traverses from the root, and searches the alphabet in the edge label one-by-one. It then finds that there is a mismatching between `AACCAA`, (the edge-label) and `ACCAA` (the sequence to be inserted). The edge is split and two new nodes are inserted. We call it *edge-splitting* in this paper. The number inside a node indicates the creation order. Note a shaded non-leaf node indicates that a suffix terminates at this node.[2]

3.1.1 Three Issues: Edge Splitting, Random Access and Data Skew

In this section, we identify three issues with the static pre-partitioning technique used in `PrePar-Suffix`.

[2] Hunt et all also suggested using bin-packing in [14] as a way to partition large sequence. They did not show the experimental results. In a private conversation, Hunt indicated that bin-packing is no better than the simple pre-partitioning technique.

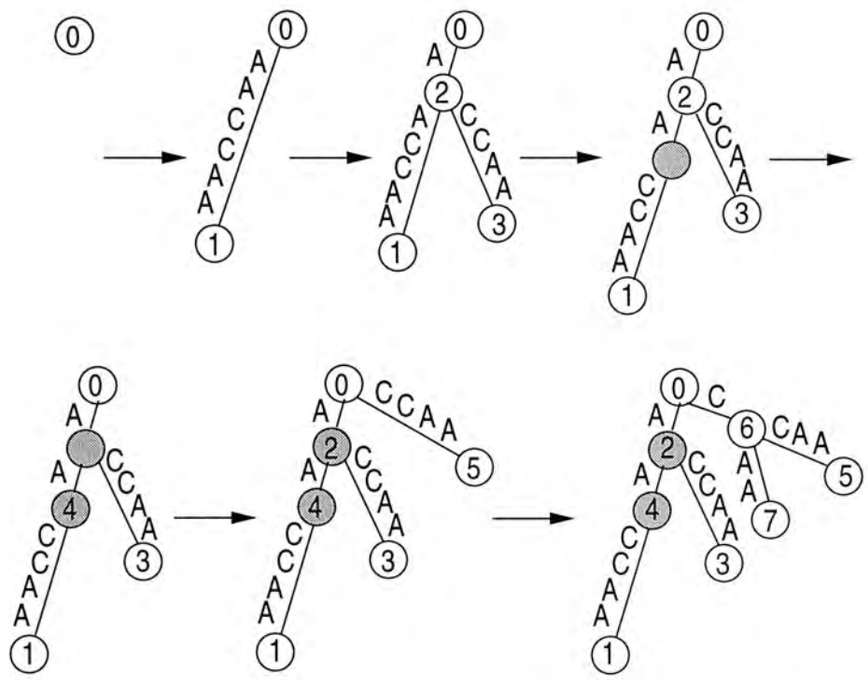


Figure 3.1: Construct a suffix tree for `AACCAA`

Edge Splitting

`PrePar-Suffix` inserts nodes, in creation order, into disk-pages sequentially. When a page is filled up, nodes will be inserted into the next page, and so on so forth. The sequential node insertion causes random disk-page accesses severely. We show a simple example to illustrate the cause of random disk-page accesses, and will analyze its consequences later. Assume that there is a subsequence `AA` from node n_i to node n_j, $n_i \rightarrow n_j$, in Figure 3.2. Then, suppose a new subsequence `AC` is inserted. Because, both `AA` and `AC` share the first letter as the common prefix, a new node n_k will be inserted in between n_i and n_j. Then `A` is associated with the edge $n_i \rightarrow n_k$ and $n_k \rightarrow n_j$ individually. Also, another node n_l is created, and `C` is associated with the edge $n_k \rightarrow n_l$. It is important to know that nodes n_i and n_j are created at the same time, and nodes n_k and n_l are created sometime later simultaneously. Therefore, the first two may be placed in the same disk-page or in two adjacent disk-pages, and the last two may also be placed in the same disk-page or in two adjacent disk-pages. As a result, more disk-pages are requested to be accessed when

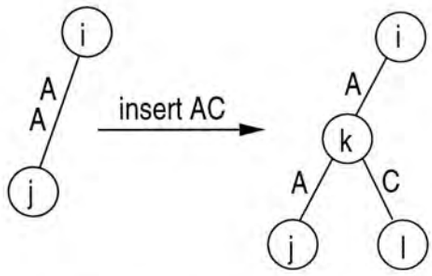


Figure 3.2: Edge-Splitting

traversing from node n_i to node n_j, because now it must traverse the page where n_k resides.

Consider the example suffix tree shown in Figure 3.1. The numbers inside the nodes indicate the creation order. The nodes, 2, 4 and 6 are non-leaf nodes and are the results of edge splitting. Suppose a single disk page can hold up to 2 nodes. The final suffix tree is stored in four disk-pages: $p_1 = (0, 1)$, $p_2 = (2, 3)$, $p_3 = (4, 5)$ and $p_4 = (6, 7)$. Initially, we can traverse from node 0 to node 1 using only one disk-page access (p_1). At the end, we need to access up to four disk-pages p_1, p_2, p_3 followed by p_1 to traverse from node 0 to node 1. It makes suffix-tree traverses highly inefficient.

Random Access

In [14], Hunt et al mentioned that `PrePar-Suffix` has the behavior of $O(n \log n)$ according to the asymptotic properties of suffix trees given by Szpankowski in [22]. As indicated in [24, 22], given a sequence, the bottleneck of insertion positions can be identified using the asymptotic properties of suffix trees. We introduce it in brief for the following discussions. A sequence of length n over a set of finite alphabets Σ can be defined as a stationary and ergodic information source that generates $\{s_k\}_{k=1}^{\infty}$, for $k = 1, 2, 3, \cdots$. Let l_k be the smallest positive number such that $s_k^{l_k-1} \neq s_i^{l_k-1}$, for $1 \leq i < k$. In other words, $l_k - 1$ is the longest length of the prefix that matches the the

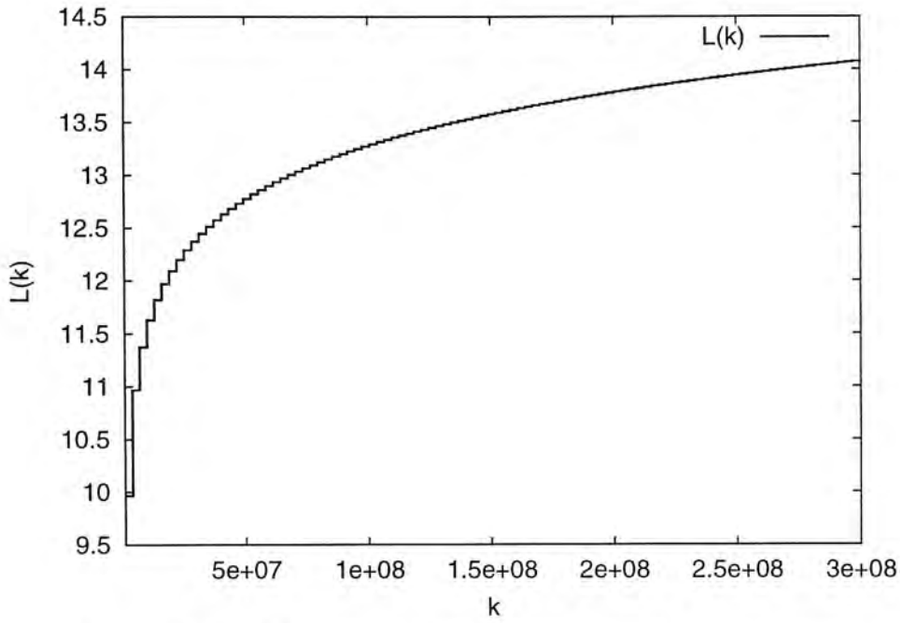


Figure 3.3: The asymptotic property of suffix trees

prefix $s_k^{l_k-1}$ before s_k. According to [24],

$$\lim_{k\to\infty} \frac{\log k}{l_k} = H$$

where H is the entropy of sequence, and the entropy of a source is defined as

$$H \equiv \sum_{x\in\Sigma} P(x) \log_2 \frac{1}{P(x)}$$

Here, $P(x)$ is the probability that x appears in the set of alphabets Σ. Under the assumption of the pseudo-random nature of DNA, $P(x) = 0.25$ for every x in {`A`, `C`, `G`, `T`}, and $H = 2$. `PrePar-Suffix` inserts a suffix using matching-and-edge-splitting technique. For a sufficiently large k, a node will be inserted at a position of l_k letters from the root. Figure 3.3 shows the relationship between suffix number k for s_k and their expected insertion position l_k. In practice, a single chromosome is ranged from 20 Mbps to 300 Mbps. Based on Figure 3.3, most insertion positions (l_k) are in the range of 12 and 15. The fact that l_k is in the range $[12, 15]$ implies that the nodes will be inserted, as

the causes of edge-splitting, in the range of level, [12, 15], in the corresponding suffix tree. A node in a suffix tree may have up to four children for `A`, `C`, `G` and `T`. Consider a full suffix-tree in the worst case. The number of nodes in that range becomes $4^{12} + 4^{13} + 4^{14} + 4^{15} = 1,426,063,360$. Suppose a 8KB disk-page can store 200 nodes. The number of 8KB disk-pages, that will be affected, is 7,130,316. As a result, due to node insertion as the causes of edge-splitting, random disk page accesses occur in a wide range of disk pages for long DNA sequences, which is substantial.

Data Skew

One basic assumption in `PrePar-Suffix` is that letters are pseudo-randomly (uniformly) distributed in a sequence. `PrePar-Suffix` pre-partitions a sequence into m-partitions evenly. We observe that data skew occurs using the pre-partitioning technique. We extract the first 10Mbps from human chromosome `Chromosome-1`. Assume a 1,150 8KB-size page buffer in memory. Then, the prefix length becomes $l = 3$, and there are totally $m = 64$ partitions. However, Figure 3.4 (a) shows that nearly a half of the 64 partitions are larger than the expected partition size (1,150 8KB pages). Figure 3.4 (b) shows I/O costs in terms of the average number of disk-page accesses using the 1,150 8KB-page buffer. Here, a LRU page replacement strategy is used in the buffer. There does not exist any extra disk-page access for the partitions that can fully fit in memory, except for the initial disk reading cost. However, once a partition cannot fully reside in the buffer in memory, extra disk-page access is unavoidable. Accessing the nodes through parent-child pointers among nodes that are randomly distributed among the pages can cause a huge number of disk-page access. Even though only a few partitions whose real size is greater than 1,550 8KB-pages (Figure 3.4 (a)), they dominate the I/O cost (over 140,000 8KB page accesses) as shown in Figure 3.4 (b).

Assume that the size of a partition is normally distributed around its mean. Figure 3.5 shows the bell shaped curve for the probability density

function of a partition size, with an expected size of 1,042 blocks and a standard deviation of 445 blocks, based on Figure 3.4 (a). The vertical line is the buffer size (1,150 8KB-size pages). Any partition that exceeds the buffer size (on the right of the vertical line) will cause extra disk-page accesses. The area bounded by the curve and the vertical line on the right is the probability that a partition will exceed the buffer size. A large number of partitions will become slightly larger than the buffer size, and a few partitions request 300 more 8KB pages than the buffer size (1150). However, the small number of partitions will dominate the I/O cost. As shown later in our extensive experimental studies, pre-partitioning techniques cannot handle data skew.

3.2 `DynaCluster-Suffix`: A New Novel Disk-Based Suffix-Tree Construction Algorithm

In this section, we show a new novel disk-based suffix-tree construction algorithm, called `DynaCluster-Suffix` (for dynamic clustering). Instead of using static pre-partitioning technique, we use a dynamic clustering technique. `DynaCluster-Suffix` constructs a disk-based suffix-tree in a cluster basis. The main advantage is that the working space is small. `DynaCluster-Suffix` can reduce both CPU cost and I/O cost, and construct disk-based suffix trees, for up to over 200Mbps DNA sequences, using only 16MB memory. As shown later, `DynaCluster-Suffix` significantly outperforms `PrePar-Suffix`.

Given a sequence of length n: $s_1 s_2 \cdots s_n$. The *prefix code* of a prefix of sufficient length l, $s_i^l = s_i s_{i+1} \cdots s_{i+l-1}$, is calculated by

$$\mathcal{P}(s_i^l) = \begin{cases} \sum_{j=0}^{l-1} c(s_{i+j}) \cdot |\Sigma|^{l-j-1} & \text{for } i \leq n-l+1 \\ -1 & \text{for } i > n-l+1 \end{cases} \tag{3.2}$$

Here, $\mathcal{P}(s_i^l) = -1$ implies that s_i will be handled as a leaf node. Given a prefix length l, there are $|\Sigma|^l$ predetermined prefixes. We use L_i for $1 \leq i \leq |\Sigma|^l$ to denote a *prefix pattern*. Any prefix of a suffix, s_k^l, must be one of the

prefix patterns. When $l = 2$, Table 3.1 shows 16 prefix patterns. We further assume that all prefix patterns are sorted in lexicographic order such as L_1 = `AA`, L_2 = `AC`, and so on so forth.

Like `PrePar-Suffix`, we dispose the use of suffix links. Therefore, the sole traverse permitted is via parent/child pointers. Unlike `PrePar-Suffix`, we explicitly define a *cluster* of nodes, and adopt a depth-first approach to create clusters recursively, while constructing a suffix tree. Here, informally, a cluster forms a small portion of a suffix tree that fits into a few disk pages, as a *working space.* Nodes in a cluster are frequently referenced to each other, and nodes across two clusters are infrequently referenced to each other. The size of a cluster is defined as $\sum_{j=0}^{l} |\Sigma|^j$. Given $|\Sigma| = 4$ for DNA. Suppose $l = 4$. Then a cluster can keep up to 341 (= 1+4+16+64+256) nodes. When the disk-page size is 8KB, and a suffix-tree node is 44 bytes, a cluster can be kept in two consecutive disk-pages.[3] Along with a cluster, an data structure is associated, called *prefix-queue* table. The prefix-queue table keeps a queue for a prefix pattern L_i consisting of all the unprocessed suffix numbers for s_k that shares the prefix pattern L_i such that $s_k^l = L_i$.

We outline our `DynaCluster-Suffix` algorithm below. Initially, we create the *root cluster*, which handles the prefix patterns that really exist in the sequence S. We insert nodes in the root cluster, while scanning the sequence. At the same time, all suffix numbers will be appended into a corresponding prefix queue in the prefix-queue table for later use. The prefix queues help to reduce the cost of scanning the whole sequence. Next, we process the prefix queue one-by-one. Let q_i be the prefix queue for a prefix pattern, L_i, picked up next. If the length of q_i is greater than or equal to a threshold τ, we create a *nonleaf cluster* for the same prefix pattern L_i at the next level. Otherwise, we create a *leaf cluster* for the prefix pattern L_i. The nonleaf cluster will have children clusters. We repeat the same procedure for the next cluster in a depth-first order, recursively. It is important to know that we do not need

[3]We can possibly choose a large disk-page. In order to make comparison with `PrePar-Suffix` later, we choose 8KB disk-pages, because the large disk-page size will reduce the number of pages in a buffer.

to scan the whole sequence again in order to append the suffix numbers into the prefix-queue table, because we can project them from the prefix-queue table for its parent cluster. Here, we call the root cluster a cluster at level 0. The children clusters of the root are at level 1, and so on so forth. We call it cluster level.

An Example

We explain the `DynaCluster-Suffix` algorithm using the same sample sequence. Consider the same sequence `AACCAA`. The length of the sequence is $n = 6$. Let the prefix length be $l = 1$. There are four prefix patterns: $L_1 =$ `A`, $L_2 =$ `C`, $L_3 =$ `G` and $L_4 =$ `T`. A cluster contains up to $\sum_{j=0}^{l} |\Sigma|^l = 1+4 = 5$ nodes. We also use a threshold $\tau = 2$ for this example.

Figure 3.6 shows the way of creating the first two clusters. In Figure 3.6, a single circle node indicates that it will be created in a disk-page. The number showed in a single circle node is the creation order (or its node identifier). A double circle node is a temporary node, that will not be stored in a disk-page. The double circle node serves a connection to the cluster at next level. The number showed in a double circle node shows the order of creation of clusters at the next level (depth-first order). Figure 3.6 (a) shows the root cluster, and Figure 3.6 (b) shows the prefix-queue table for the root cluster. The root node is numbered 0, and has four possible children for `A`, `C`, `G` and `T`. Because there are no `G` and `T` in this sequence, we only create two nodes for `A` and `C` (Figure 3.6 (a)). The node creation for the root cluster is explained below. Recall the prefix length for this example is $l = 1$. We scan the sequence from left to right. First, we create the root node and number it as 0. Second, we handle the first suffix $s_1 =$ `AACCAA`. Its prefix s_1^1 is `A`. Because `A` does not appear as an edge-label in the root cluster, we create the leftmost double circle with a number 0. `A` becomes the edge-label along the newly created edge. At the same time, the suffix number 1 for the first suffix s_1 is appended into the corresponding prefix queue in the prefix-queue table (Figure 3.6 (b)). Third, we handle the second suffix $s_2 =$ `ACCAA`. Its

s_2^1 is A too. The suffix number 2 is appended into the corresponding prefix queue (Figure 3.6 (b)). Because, A is already inserted into the root cluster, we do not need to insert it again into the root cluster. Fourth, we handle the third suffix s_3 = CCAA. Its prefix s_3^1 is C. Because C does not appear as an edge-label in the root cluster, we create the rightmost double circle with a number 1. C becomes the edge-label along the newly created edge. At the same time, the suffix number 3 for the third suffix s_3 is appended into the corresponding prefix queue in the prefix-queue table (Figure 3.6 (b)). We repeat it for every alphabet in the sequence from left to right. Finally, after finished scanning, the root cluster and its corresponding prefix queues are shown in Figure 3.6 (a) and (b).

The root cluster is the only cluster at level 0. Next, we create clusters at the next level following the numbers shown in the double circle nodes (Figure 3.6 (a)). We create a cluster at level 1 following the depth-first order. It is important to know that, at level 0, the length of the prefix queue for the prefix pattern L_1 (A) is $|L_1| = 4$ and the length of the prefix queue for the prefix pattern L_2 (C) is $|L_2| = 2$. Because $\tau = 2$, we will create a nonleaf cluster at level 1 for L_1, and a leaf cluster at level 1 for L_2.

Figure 3.6 (c) shows the the first cluster being created at level 1 (in the dotted rectangle). We first create the root node for the cluster, and number it as 1. The root node will be stored in a disk page. It is important to know that a cluster is created according to a prefix queue, q_i, in its parent cluster. In other words, all sequences created from the cluster and beyond share the common prefix pattern L_i. The cluster in Figure 3.6 (c) is created for the prefix queue, q_1, for the prefix pattern L_1 (A) in Figure 3.6 (b). As can be seen in Figure 3.6 (b), the prefix queue q_1 consists of four suffix numbers: 1, 2, 5 and 6, for the four suffixes: s_1, s_2, s_5 and s_6 in order. For any suffix s_k in the prefix queue q_1 in its parent cluster, we process the $s_{k+j \cdot l}$ suffix where $j = 1$ is the cluster level. First, for s_1, we process $s_2 = s_{1+1 \cdot 1} =$ ACCAA. Its prefix is s_2^1 = A. Because A does not appear as an edge-label in the current cluster, we create the leftmost double circle with a number 0 (in the

dotted rectangle in Figure 3.6 (c)). A becomes the edge-label along the newly created edge. At the same time, the suffix number 1 is appended into prefix queue corresponding to the prefix pattern A in the prefix-queue table (Figure 3.6 (d)). Second, for s_2, we process $s_3 = s_{2+1\cdot1} =$ CCAA. Its prefix is $s_3^1 =$ C. Because C does not appear as an edge-label in the current cluster, we create the rightmost double circle with a number 1 (in the dotted rectangle in Figure 3.6 (c)). C becomes the edge-label along the newly created edge. At the same time, the suffix number 2 is appended into the prefix queue corresponding to the prefix pattern C in the prefix-queue table (Figure 3.6 (d)). Third, for s_5, we process $s_6 = s_{5+1\cdot1} =$ A. Its prefix is $s_5^1 =$ A. The suffix number 5 is appended into the prefix queue corresponding to the prefix pattern A (Figure 3.6 (d)). There is no need to insert it into the cluster, because A has already appeared in the cluster. Finally, for s_6, because it is the last suffix, we insert it into the cluster, and indicate it using a shaded node which means a suffix terminates at this nonleaf node. We do not need to append s_6 into any prefix queue.

It is worth noting that two prefix queues for the prefix patterns A and C in Figure 3.6 (d) consist of less than $\tau = 2$ suffixes. Therefore, we create two leaf clusters. The three leaf clusters are shown as dotted rectangles in Figure 3.7. The left and right leaf clusters are created as a new cluster. For the middle leaf cluster, we do not need to create a new cluster because it only has one possible suffix to be handled. We replace the corresponding temporary node (double-circle node) for this suffix.

Algorithm

The `DynaCluster-Suffix` algorithm is shown in Algorithm 2 and Algorithm 3. The main body of the algorithm is between line 1 - 4 in Algorithm 2. It first calls the rootCluster procedure to create the root cluster, and then calls the nextClsuter procedure to create clusters at the next level. The procedure rootCluster is between line 6-17, and the procedure nextCluster is between line 19-31. As discussed above, the procedure nextCluster is a recursive pro-

cedure, and will call either nonleafCluster procedure or leafCluster procedure depending on the length of a prefix queue. The nonleafCluster procedure is between line 1-13, and the leafCluster procedure is between line 15-26 in Algorithm 3.

Algorithm 2 DynaCluster-Suffix

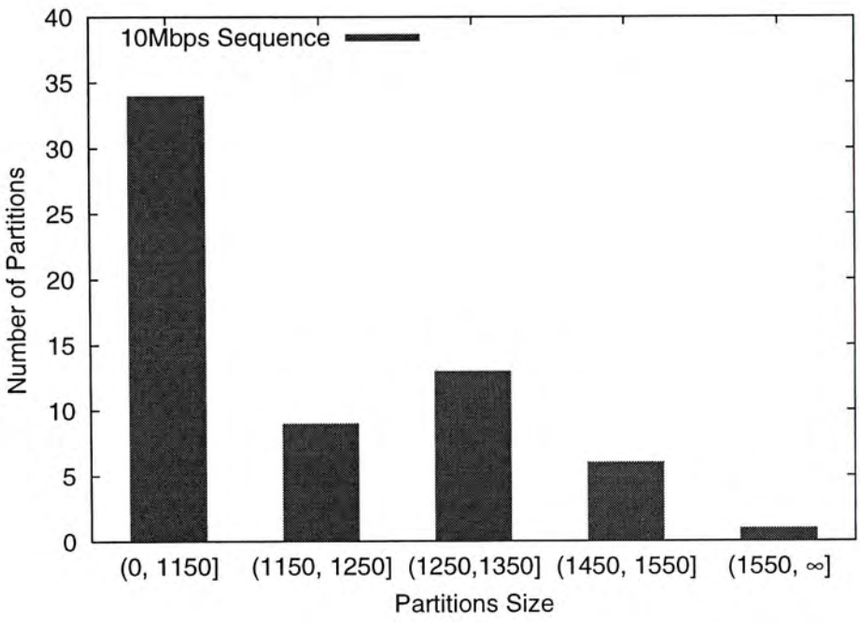
Input: a sequence of length n, S, a sufficient prefix length l, and a threshold τ.
Output: a disk-based suffix tree, X.
Let the sequence S of length n be $s_1 s_2 \cdots s_n$. Let X_j and Q_j be a cluster and its prefix-queue table at level j.

```
1: begin
2: (X_0, Q_0) ← rootCluster(S, n, l);
3: nextCluster(X_0, Q_0, 0, l);
4: end
5:
6: procedure rootCluster(S, n, l)
7: begin
8: create the root node for the cluster X_0 for the suffix tree, and create Q_0 for prefix
   queues.
9: for every suffix s_k do
10:   compute the prefix code P(s_i^l);
11:   if s_k^l does not exist in X_0 then
12:     insert the prefix s_k^l into X_0;
13:   end if
14:   insert the suffix number k into the prefix queue for the prefix pattern s_k^l in Q_0;
15: end for
16: return (X_0, Q_0);
17: end
18:
19: procedure nextCluster(X_j, Q_j, j, l)
20: begin
21: for every prefix queue q in Q_j do
22:   if length(q) ≤ τ then
23:     leafCluster(x_q, q, j, l); {let x_q be the leaf node in X_j corresponding to q.}
24:   else
25:     (X_{j+1}, Q_{j+1}) ← nonleafCluster(x_q, q, j, l); {let x_q be the leaf node in X_j corre-
        sponding to q.}
26:     for ever prefix queue q' in Q_{j+1} do
27:       nextCluster(x_{q'}, q', j + 1, l); {let x_{q'} be the leaf node in X_j+1 corresponding
          to q.}
28:     end for
29:   end if
30: end for
31: end
```

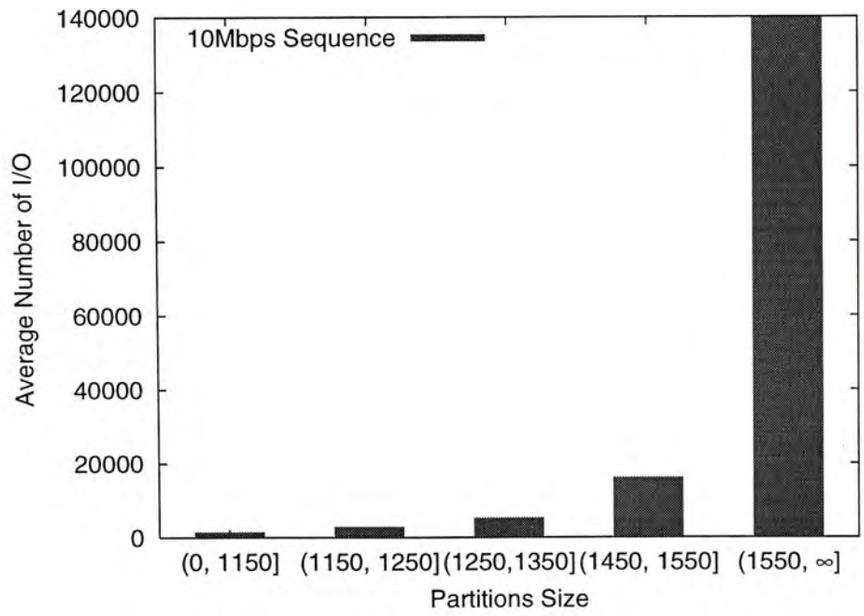
Algorithm 3 `DynaCluster-Suffix`

procedure nonleafCluster(x_q, q, j, l)
begin
create the root node for the cluster X_{j+1}, and create Q_{j+1} for the possible prefix queues;
for every suffix s_k in q **do**
  compute the prefix code $\mathcal{P}(s^l_{k+j\cdot l})$
  if p_i does not exist **then**
    insert the prefix $s^l_{k+j\cdot l}$ into X_{j+1};
  end if
  insert the suffix number k into the prefix queue for the prefix pattern $s^l_{k+j\cdot l}$ in Q_{j+1};
end for
connect the root of X_{j+1} with x_q;
return (X_{j+1}, Q_{j+1});
end

procedure leafCluster(x_q, q, j, l)
begin
if $length(q) = 1$ **then**
  make x_q as a leaf node;
else
  create a cluster X_{j+1};
  for every suffix s_k in q **do**
    insert the suffix $s_{k+j\cdot l}$ into X_{j+1};
  end for
  connect the root of X_{j+1} with x_q;
end if
end



(a) Partition Size Distribution



(b) I/O and Partition Size

Figure 3.4: Partition size and I/O cost for the first 10 Mbps of `Chromosome-1`

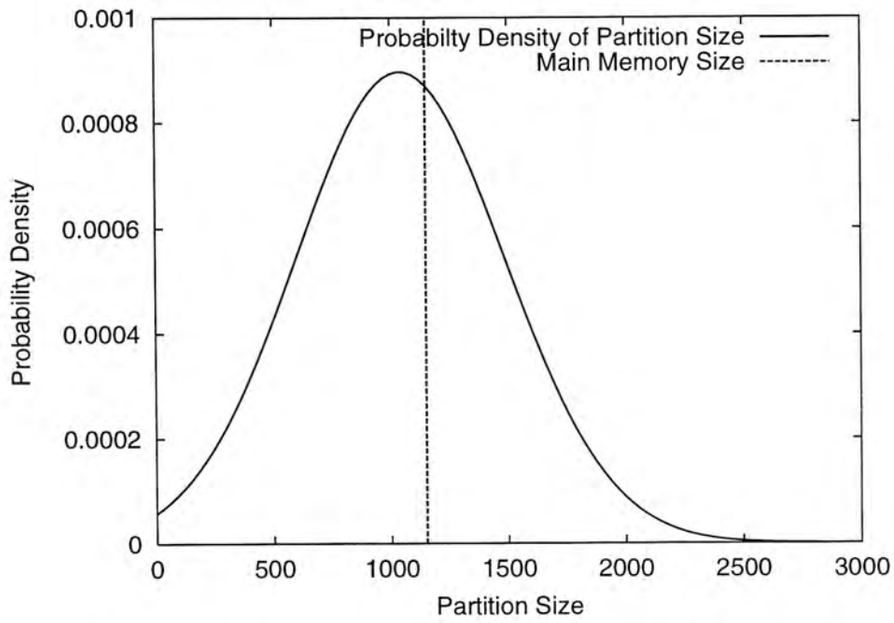
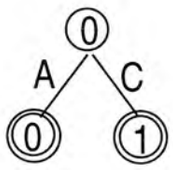


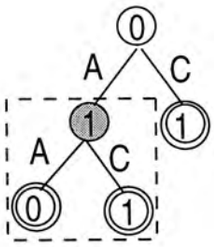
Figure 3.5: Probability density of partition sizes



(a)

Prefix Pattern	Prefix Code	Prefix Queue
A	0	1, 2, 5, 6
C	1	3, 4
G	2	-
T	3	-

(b)



(c)

Prefix Pattern	Prefix Code	Prefix Queue
A	0	1, 5
C	1	2
G	2	-
T	3	-

(d)

Figure 3.6: Creation of the first two clusters for AACCAA

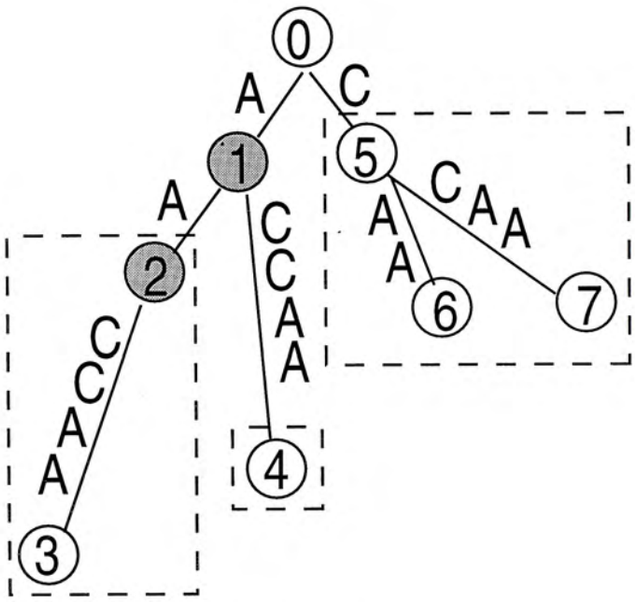


Figure 3.7: The disk-based suffix tree for `AACCAA`

Chapter 4

Suffix Links Rebuilt

In the linear time in memory suffix trees constructions, suffix-links are often used to speed up the construction [11, 12, 20]. In addition to the construction, suffix-links are also used in finding repetitive structures in a sequence in linear time [12] and approximate sequence matching. We will discuss how to rebuild the suffix-links of a disk-based suffix tree using the constant time LCA (least common ancestors).

4.1 Suffix-links and Least Common Ancestors

It is well known that the least common ancestor of two nodes x and y can be found in constant time after linear time and space preprocessing [13, 21]. In [7], Farach et al pointed out a possible way to construct suffix links. But they did not provide any mechanisms to support this notion. There is no reported study on rebuilding suffix-links for disk-based suffix trees. In this section, we show how we rebuild the suffix links for the disk-based suffix tree. Base on Farach et al observation, we rebuild suffix links as an application of *lca*-queries (least common ancestor queries). Figure 4.1 illustrates an example, where a is an alphabet, and α, β and γ are arbitrary subsequences of S. Let the two leaf nodes, s and t, represent two suffixes: $a\alpha\beta$ and $a\alpha\gamma$ such that

$\sigma(s) = a\alpha\beta$ and $\sigma(t) = a\alpha\gamma$. If node s and v have a least common ancestor at node v, then, $a\alpha$ $(= \sigma(v))$ is the longest common prefix of the two sequences: $a\alpha\beta$ and $a\alpha\gamma$. There is a suffix link from node v to w such that $sl(v) = w$, iff $\sigma(v) = a\sigma(w)$. It is important to know that, if $sl(v) = w$, each leaf node s, in the subtree rooted at v, must be able to find a corresponding node x such that $\sigma(s) = a\sigma(x)$ where $\sigma(s) = s_i$ and $\sigma(x) = s_{i+1}$. Recall any internal node has at least two children. Therefore, if we know the two smallest suffixes, s_i and s_j in a subtree, then we can quickly compute s_{i+1} and s_{j+1}. Let node s, t, x and y represent the suffixes s_i, s_j, s_{i+1}, s_{j+1}, then $sl(lca(s,t)) = lca(x,y)$.

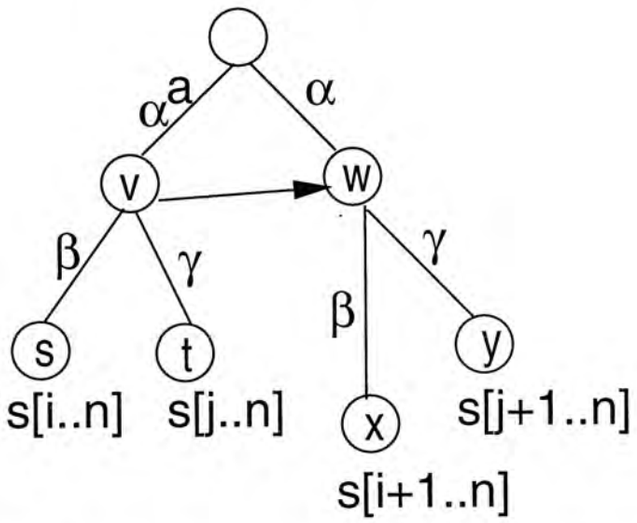


Figure 4.1: Suffix links and least common ancestors

Here, we treat suffix-link rebuilt as an application of *lca*-queries. Several fast algorithms for finding least common ancestors were proposed [13, 21]. All these algorithms are of in-memory algorithms. There are no reported studies on how to find least common ancestors where the tree cannot fit in memory. There are no reported studies on how to use this technique to rebuild suffix-links for suffix trees for long DNA sequences. We extended Gusfield's implementation[1] of the fast algorithm for finding the least common ancestors [21]. As shown in [21], in order to answer *lca*-queries using constant time

[1] `http://www.cs.ucdavis.edu/~gusfield/strmat.html`

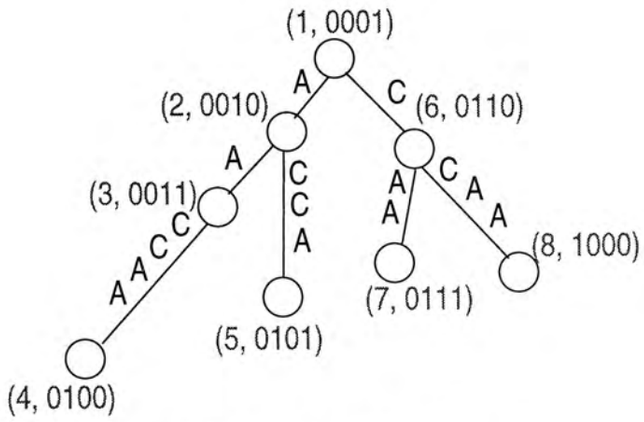
$O(1)$, a preparation phase is needed in which information such as `preorder` number, `inlabel` number, `ascendant` number, etc, needs to be collected. It is important to know that the preparation phase requests to traverse the disk-based suffix tree twice, using depth first traversal order. The outcome of the tree traversals is a table, called *lca*-table, in which the information needs for answering *lca* queries is maintained. In *lca*-table, for a single alphabet in the sequence, 12 bytes are needed.

In the following, in order to show the reasons why we need to traverse the tree at least twice, we show how the preparation phase computes `preorder` number, `inlabel` number and `ascendant` number. For detail, refer to [13, 21, 11]. In order to assign a `preorder` number to every node in the suffix tree, it needs to traverse the suffix tree using depth-first search. Figure 4.2 (a) shows `preorder` numbers for the suffix tree for `AACCAA`, where both decimal `preorder` number and binary `preorder` numbers are shown as a pair for each node. `Inlabel` number of a node, v, is a `preorder` number in the subtree rooted at v that has the maximal number of rightmost "0" bits (Figure 4.2 (b)). Both `preorder` and `inlabel` numbers can be obtained in one depth-first traverse. After obtained `inlabel` numbers, it needs to traverse the suffix tree in the second time following the depth-first order to assign `ascendant` number to every node. An `ascendant` number is assigned as follows. Recall that the root node has the largest `inlabel` number 2^l. The `ascendant` number for the root is the same as its `inlabel` number. Consider a node v and its parent u. If `inlabel` number of v is the same as `inlabel` number of u, then `ascendant` number of u becomes the `ascendant` number for v. Otherwise, `ascendant` number of u plus 2^i becomes `ascendant` number of v, where i is the index of the rightmost "1" in `inlabel` number of v (Figure 4.2 (c)).

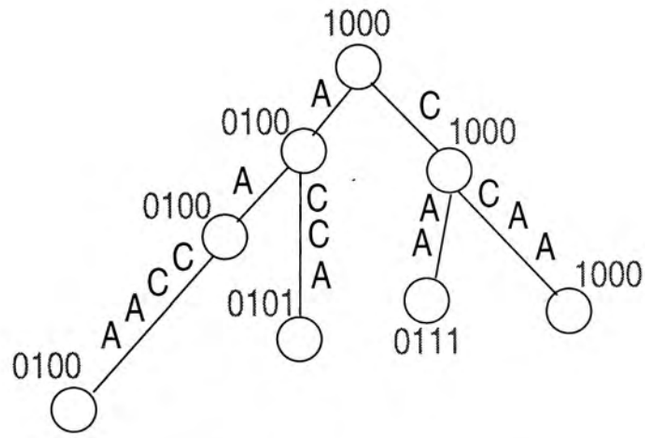
Figure 4.3 shows two data structures in addition to *lca*-table. Figure 4.3 (a) shows a `leave` data structure, the i-th entry, denoted l_i, points to the suffix s_i in the suffix tree. For example, l_1 points to s_1 (`AACCAA`), l_5 points to s_5 (`AA`), and l_6 points to the last suffix (`A`). The number associated with

a node in Figure 4.3 (a), is the `preorder` number of the node. Figure 4.3 (b) keeps two leaf nodes, l_i and l_j, for an internal node as discussed above. We call it `two-leaf` table. The `two-leaf` table as well as the *lca*-table is maintained following the `preorder` order.

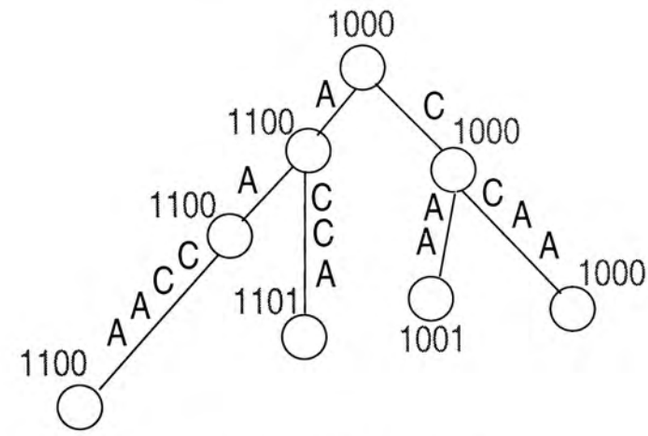
In the following, we show how to add a suffix-link for the node with a `preorder` number 3 in Figure 4.3 (a). First, for the node with the `preorder` number 3, we find its two leaf nodes, $l_i = 1$ and $l_j = 5$ in the `two-leaf` table (Figure 4.3 (b)). Second, with a simple calculation, we know we need to find the least common ancestor for $l_i + 1 = 2$ and $l_j + 1 = 6$. Third, by checking `leave` in Figure 4.3 (a), we find that the 2nd and the 6th pointers point to two nodes, with `preorder` number 3 and 2, respectively. Finally, the least common ancestor for the two numbers, 2 and 3, are calculated using *lca*-table, which can be done in constant time $O(1)$. The *lca* of the two nodes is the node with the `preorder` number 2 shown in Figure 4.3 (a). Therefore, there exists a suffix-link from node 3 to node 2.



(a) `Preorder` Number

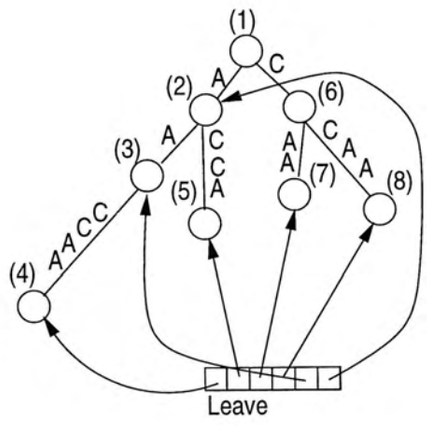


(b) `Inlabel` Number



(c) `Ascendent` Number

Figure 4.2: *lca*-query preparation



(a)

	l_i	l_j
1	1	3
2	1	2
3	1	5
4	-	-
5	-	-
6	3	4
7	-	-
8	-	-

(b)

Figure 4.3: Suffix link rebuilt for the suffix tree for `AACCAA`

Chapter 5

q-Length Exact Sequence Matching

There are a number of linear time algorithms for the q-length exact sequence matching problem. For example, the Boyer-Moore [4] and Knuth-Morris-Pratt [18] both can solve the problem in linear time. We define the exact sequence matching problem as follows: Let $S[1 \cdots n]$ and $P[1 \cdots q]$ be two sequences of length n and q respectively. The problem is to find all occurrences k such that $S[k \cdots k+q] = P[1 \cdots 1]$. Although the Boyer-Moore and Knuth-Morris-Pratt can solve the problem in linear time, solving the problem by suffix-trees are particularly advantageous when S is large and static and P is small and changing. The reason is that after constructed the suffix-tree T for S, the running time of exact sequence matching is proportional to the length of the pattern P, which is $O(q)$. Therefore, using suffix-trees for exact sequence matching in genome is efficient.

5.1 q-Length Exact Sequence Matching by Suffix-Tree

The problem of q-length exact sequence matching is solved in two stages. In the first stage, we walk down the suffix-tree to find a path that match

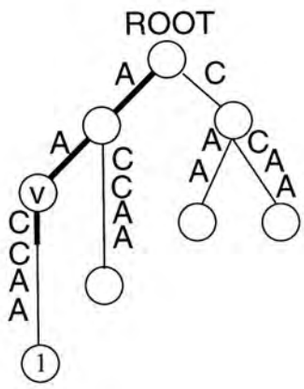


Figure 5.1: Exact String Matching using Suffix-Tree

with the pattern. If any mismatch is found, then S does not contain P and returns NOTFOUND. Otherwise, the pattern will terminate in an edge with a node v. All the suffixes under v will contain the pattern. In the second stage we traverse the subtree of v and put all suffix numbers into an array of occurrences, N. The pseudo-code is shown in algorithm 4.

Figure 5.1 illustrates the idea of q-length exact sequence matching. Suppose we want to find all occurrence of a pattern AAC in the sequence $AACCAA$. We first construct the suffix tree for the sequence (Figure 5.1) and walk down from the root to find the pattern. The path is shown in the darkened line in the tree and terminates at v. We traverse down the subtree of v and found out that suffix 1 contains the pattern.

Algorithm 4 Exact Sequence Matching

Input: a suffix-tree T for a sequence of $S[1 \cdots n]$, and a query pattern $P[1 \cdots q]$

Output: an array of occurrences N of k such that $S[k \cdots k+q] = P[1 \cdots q]$

```
1: procedure exQuery(T, P)
2: walk down from the root of T to a node v until a mismatch is found or
   q exhausted; {stage 1}
3: if No mismatch is found then
4:    traverse down the subtree of v and put all the suffix numbers into N;
      {stage 2}
5:    return N;
6: else
7:    return NOTFOUND;
8: end if
```

Chapter 6

Implementation

In this chapter, we will first give an overview of our system and outline our data structure for representing a suffix tree in a secondary storage. Then we will discuss about the paging and buffering strategies used in our implementation.

6.1 System Overview

In this section, we introduce our system and the core components. Figure 6.1 exhibits the various components in our system. A DNA sequence is stored in a file and passed to the index builder; the builder contructs a persistent suffix tree in disk. In exact sequence matching, the DNA sequence and the suffix tree index generated are passed to the query processor. The query processor processes the suffix tree and locates all the occurrences of a query pattern. In finding all tandem repeats in a sequence, the suffix links index is also required. The suffix links index regenerator takes in a DNA sequence and its suffix tree, and then populates the suffix links index using the technique of constant time lowest common ancestors. The sequence, suffix tree and suffix links index are put together into the tandem repeats finder and the tandem repeats finder will return all tandem repeats in the sequence.

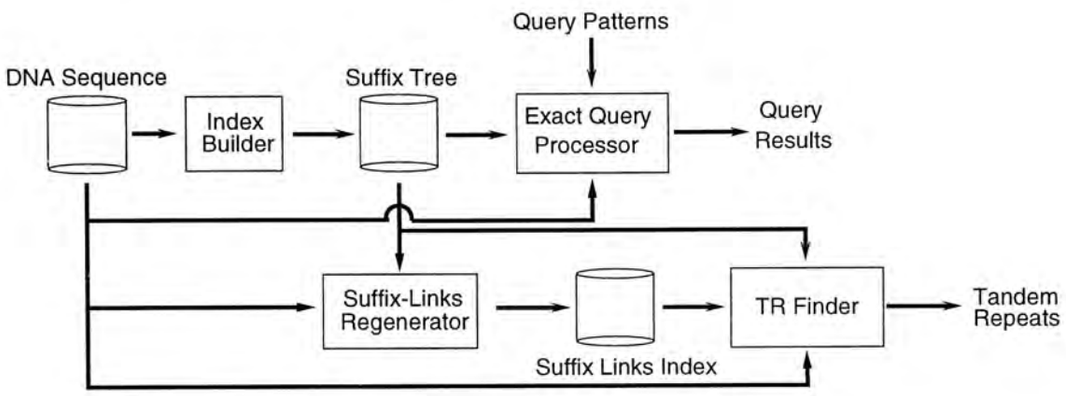


Figure 6.1: System Overview

6.1.1 Index Builder

The index builder constructs a suffix tree from a biological sequence. There are various algorithms in constructing a suffix tree from a sequence. In our implementation, we have implemented the In-memory Ukkonen's algorithm, Disk-Based Ukkonen's algorithm, PrePar-Suffix algorithm and our novel Dyna-Cluster Suffix algorithm. The Ukkonen's algorithm uses suffix links to achieve linear time performance and the suffix link index is already present immediately after construction. However, in Pre-Par suffix and our Dyna-Cluster Suffix algorithms, suffix link index is re-generated when necessary.

6.1.2 Exact Query Processor

The exact query processor takes in a number of sequence patterns, a biological sequence and a suffix tree of the sequence. Since exact sequence matching problem does not require the suffix links index, suffix trees generated by Pre-Par suffix and our Dyna-Cluster suffix algorithms can be used without suffix links regeneration. The processor directly implements the algorithm stated in Chapter 5.

6.1.3 Suffix Links Regenerator

The suffix links regenerator implements the algorithm stated in Chapter 4. Suffix links regeneration is only one application of the constant time lowest common ancestor algorithm, and there are a number of more advanced algorithms like k-difference approximate sequence matching takes the constant time LCA as their core. In preparing constant time LCA queries, we need an array of *LCA* data structure. The structure has three elements. They are the InLabel, Leader, and Ascendents. They are prepared in two distinct depth-first traversal over the suffix tree. The LCA structures can be used by others constant time LCA applications. In regenerating suffix links, we also need an array pointing to the leave node of each suffix. The two leaves array remembers the the leave numbers of any branching subtree of a node.

6.1.4 Tandem Repeats Finder

The tandem repeats finder implements Gusfield's *Linear Time Algorithms for Finding and Representing all Tandem Repeats in a string*[12]. The algorithm is one of the applications of suffix trees and suffix links.

6.2 Data Structures

In this section, we will discuss the representation of of a suffix tree. A suffix tree is made up by nodes and leaves. Figure 6.2 illustrates a simple suffix tree. Internal nodes (including root) are in grey and leaves are in white. We will discuss the representation of nodes and leaves in the following subsections.

6.2.1 Representation of a Node

A node is represented as figure 6.3 and the corresponding data members are listed in table 6.1. Every node has a unique node identifier, *id*, and the identifier is filled up after the tree construction. There is a *parent* pointer and an array of *children* pointers that points to every child node with an

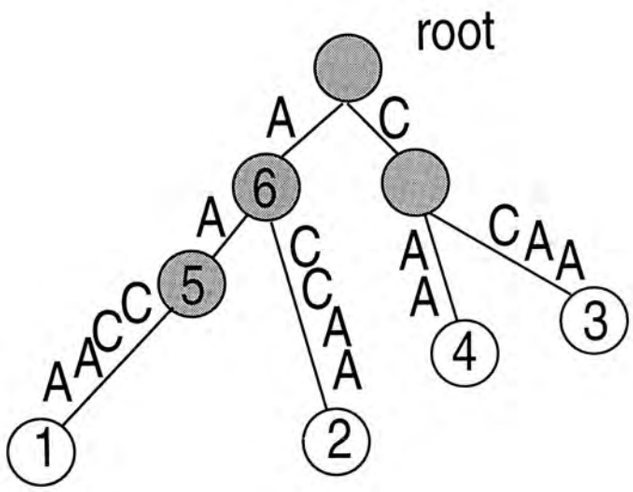


Figure 6.2: Representation of a suffix tree

Data Member	**Meanings**
id	Node identifier
parent	Pointer to the parent
children	Direct pointers to the children
edgestr	Pointer to the beginning of the edge label
edgelen	Length of the edge label
isaleaf	A flag that distinguish between a node and a leaf
intleaf	Suffix number of a node

Table 6.1: Data members of a node

alphabet in Σ. In the other words, for DNA sequences, every node has four pointers pointing to the four children of the node. When a child node does not exist, the pointer is assigned to *NULL*. The *edgestr* points to the starting position of the edge label in the sequence and the *edgelen* is the length of the edge label. The *isaleaf* is a flag that distinguishes a node from a leaf in a suffix tree and the *intleaf* stores the suffix number if there is a suffix that terminates in the node.

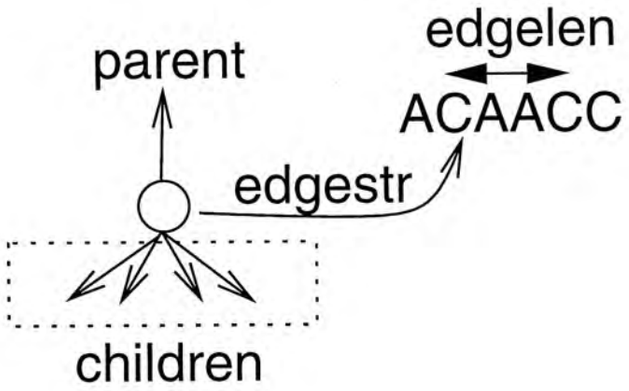


Figure 6.3: Representation of a node

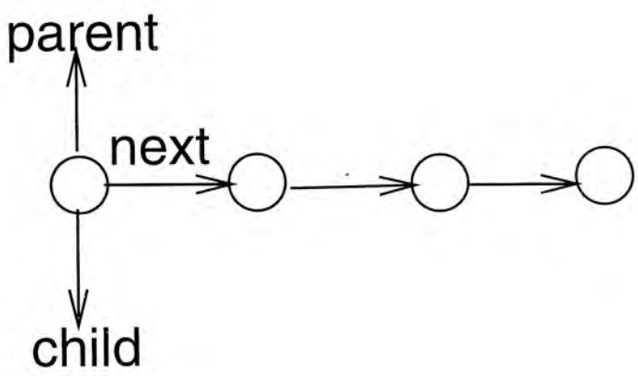


Figure 6.4: An alternative node representation

6.2.2 An Alternative Node Representation

Besides representing every child node as a direct pointer, we may link the children of a node by the *next* pointer (figure 6.4). This alternative representation is smaller in size and more adaptive to a variable alphabet size. It is recommended when the suffix tree can be fitted inside main memory. However, when the suffix tree cannot be stored in the main memory, traversing the next pointers will introduce a very large performance overhead. Suppose that we need to access the third child of a node, we first obtain the first child and get the next pointer. Then we walk to the second node and finally, from the next pointer of the second, we reach the third node. Since nodes linked

Data Member	Meanings
id	Node identifier
parent	Pointer to the parent
edgestr	Pointer to the beginning of the edge label
edgelen	Length of the edge label
isaleaf	A flag that distinguish between a node and a leaf
pos	Suffix number of the leaf

Table 6.2: Data members of a leaf

by the next pointer are not necessarily located in the same disk page, the traversal will introduce extra paging activities.

6.2.3 Representation of a Leaf

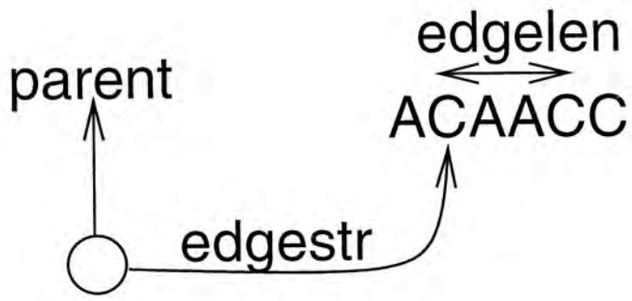


Figure 6.5: Representation of a leaf

The leaf structure is shown in figure 6.5 and the data members are in table 6.2. A leaf differs from a node becuase a leaf does not possess any child and *intleaf*, but a leaf does have a suffix number. A suffix tree for a sequence of size m has exactly m leaves and the concatentation of edge labels from the root to the leaf exactly spells out the unique suffix. The starting position of the suffix is recorded as the suffix number in a leaf.

6.3 Buffering

Since accessing data in RAM (100ns) is much faster much faster than accessing data on disk (20ms), and RAM is expensive and limited, buffering is necessary. Because of the *Locality of Reference*, programs tend to reuse data and instructions that they are recently used. It is therefore possible to map a large disk space into a smaller but faster main memory. In the following subsections, the implementation of our buffering system will be discussed.

6.3.1 Page Format

The overview of our buffering system is shown in figure 6.8. Since our suffix trees are persistent suffix trees, all the nodes and leaves are stored in disk pages. A page is the primitive unit of storage and I/O activity. In addition to store the actual data, a page also contains house keeping information. In our implementation, the page size is equal to 8K. The first integer of a page indicates the total number of records and the second integer tells the free position for futher insertion. As in figure 6.6, there are three items in the page and the *freepos* points to the next available position for further insertions. At the end of a page, there is an array of *slots* pointers. Each slot pointer points to the starting position of a record. For example, slot 1 points to the first item and slot 2 points to the second one.

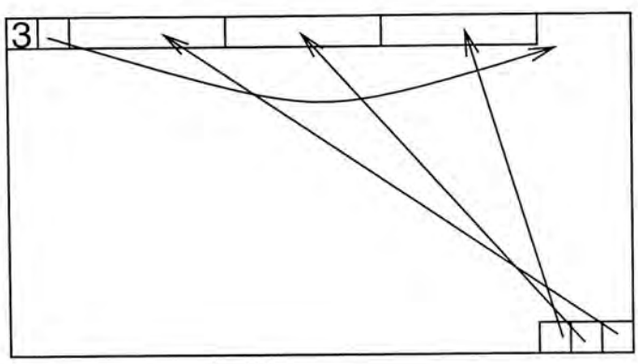


Figure 6.6: Page format

6.3.2 Address Translation

Any virtual address must be translated into physical address before use, and in our implementation, the first 22 bits of the virtual address is devoted to page number and the last 10 bits are for slots number. As a result, there are at most 4194304 pages and 1024 items in a page. Given an average of 55 bytes per character, we can construct an index for 624 Mbps sequences at most. Figure 6.7 shows the construction of virtual address.

From the virtual address, we obtain the page number. The page number is translated into the physical address in memory. We look up the physical address from the page table. A page table is simply an array of pointers pointing to the buffer memory occupied by the page identified by the page number . If the page does not reside in the buffer, we have to swap out a page in the buffer and replace it with the requested page from disk. Otherwise, the page resides in buffer. After we looked up the page number in page table, we got the physical memory address of the requested page. We calucate the position of item in the page by looking up the slots array in the page. Finally, we return the exact address of the item in the buffer.

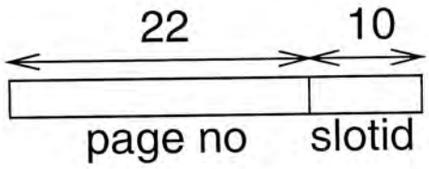


Figure 6.7: Virtual address

6.3.3 Page Replacement Strategies

A page replacement occurs when we swap out a page in buffer and replace it with our requested page. There are various page replacement strategies. We outline the strategies as follows.

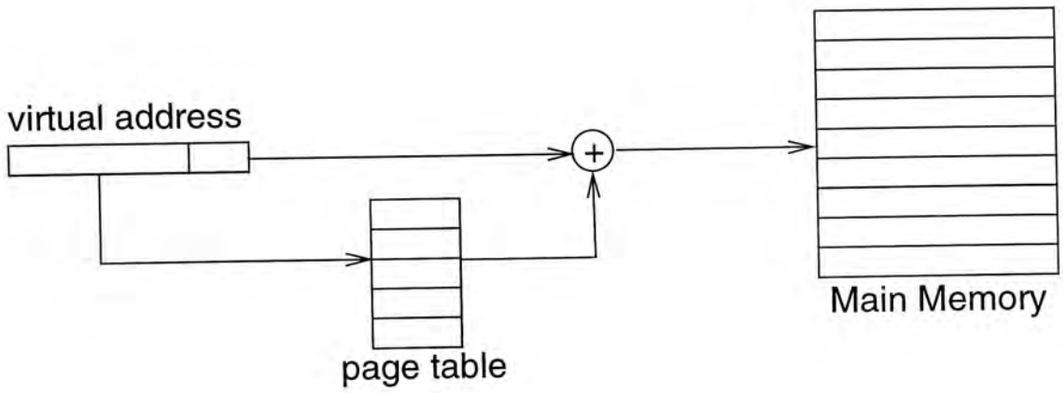


Figure 6.8: Virtual address translation

1. *Random Replacement* strategy randomly picks up a page. It is the easiest to implement but also obviously not optimal.

2. *First-In, First-Out* (FIFO) strategy replaces the pages in a circular fashion and maintains a circular buffer list. A *next pointer* to the list is held and it points to the page to be released. As a new page is swapped in, the pointer increments to the next page. As a result, the next page to be replaced will be the oldest in the buffer. It is easy to implement but the strategy takes no account of how often a page is used. It may replace a page which is being used repeatedly but stays for a long time.

3. *Least Recently Used* (LRU) strategy is widely used because it follows the *Locality of References*. It picks up a victum page which has not been referenced for the longest time. However, LRU is seldom used directly because of its high overhead. To use the LRU strategy, a counter must be used for every page in buffer and the counter is updated every time the page is referenced. To locate the victum, we have to search from the counters and it may be very expensive.

4. *Clock* Strategy is a modification of the FIFO strategy. Since FIFO does not take the reuse frequency, the clock strategy adds an *use* bit to every page in buffer. The use bit keeps track whether the page has been used

recently. It is set to 1 when the page is accessed. A circular buffer list and a *next pointer* to next page are held as the FIFO. When the system is looking for a page to be replaced , the list is traversed. If the page pointed by the pointer has the *use* bit set, then the bit is cleared and we increment the next pointer. When a page is found with the *use* bit unset, the page has not been recently used and it is replaced to make room for the new page.

After studied the performance of LRU and the Clock approaches, we found that the Clock approach has a reasonable I/O performance and a lower overhead than the LRU. Consequently, we choose to use the Clock strategy.

Chapter 7

A Performance Studies

We conducted an extensive performance study using a Pentium-4 2GHz CPU PC running Red Hat Linux 7.2 with 1GB main memory. All testing are done on top of a prototype system using buffer management with LRU (Least Recent Use) strategy. All disk-pages are formatted pages where each page has a `header` (containing control information) and a slot for each data record in the page. The page-number and the slot number together serve the disk-based node identifier for each suffix tree node. All disk-pages are of 8KB pages. No locking and logging are used in our testing. We implemented `PrePar-Suffix` and `DynaCluster-Suffix` as well as our prototype system using `g++` 2.96. The notations and definitions, together with the default values, for all the parameters are summarized in Table 7.1.

Notation	**Definition (Default Values)**
$\|\Sigma\|$	the size of the alphabet for DNA (4)
n	the length of the sequence S
d	the number of bytes for an alphabet in S (60)
M_S	the main memory for holding D in memory ($n \times d$)
M_A	the size of main memory available (MB) (16)
l_{pp}	the prefix length used for `PrePar`
m	the number of partitions used in `PrePar-Suffix`
l_{df}	the prefix length used for `DynaCluster` (4)
τ	the threshold used in `DynaCluster` (1024)

Table 7.1: System parameters.

In our extensive performance studies, we compare our DynaCluster with PrePar-1 (no partitioning) [15], PrePar (partitioning a sequence into m partitions) [14], the in-memory version of Ukkonen's algorithm [5, 11], denoted Ukkonen-M, implemented by Gusfield in his strmat package[1], and our implementation of disk-based version of Ukkonen's algorithm, denoted Ukkonen-D. We reinforce that, in strmat, Ukkonen-M runs purely in memory and does not use any buffering facility. Instead, Ukkonen-D allows a suffix tree to be built beyond the amount of main memory available. There are several points we need to make upon the use of main memory, and some important parameters.

- All disk-based suffix-tree construction algorithms, Ukkonen-D, PrePar and DynaCluster we have implemented using the exact same data structure for disk-based suffix tree, and the same disk-page format.
- Ukkonen-D constructs a disk-based suffix tree with suffix links at the same time.
- Both DynaCluster and PrePar (as well as PrePar-1) first construct a disk-based suffix tree without suffix links. Then, suffix-links are rebuilt using our *lca* technique. Upon the construction of disk-based suffix-tree without suffix links, DynaCluster divides the main memory available (M_A) into two parts. One part is for the suffix-tree and the other is for keeping prefix queues (suffix numbers). The main memory for the suffix-tree part is fixed to 8MB for all the testing. The rest of main memory ($M_A - 8$)MB is used for keeping prefix queues. Two separated buffers are used. The reason behind is that DynaCluster needs a very small working space when handling clusters of nodes when construction, and it is more beneficial to keep the prefix queues in memory. On the other hand, PrePar (PrePar-1) uses all memory space M_A for keeping the suffix tree to be built.
- In order to focus on the effectiveness of buffer behaviors, we keep the

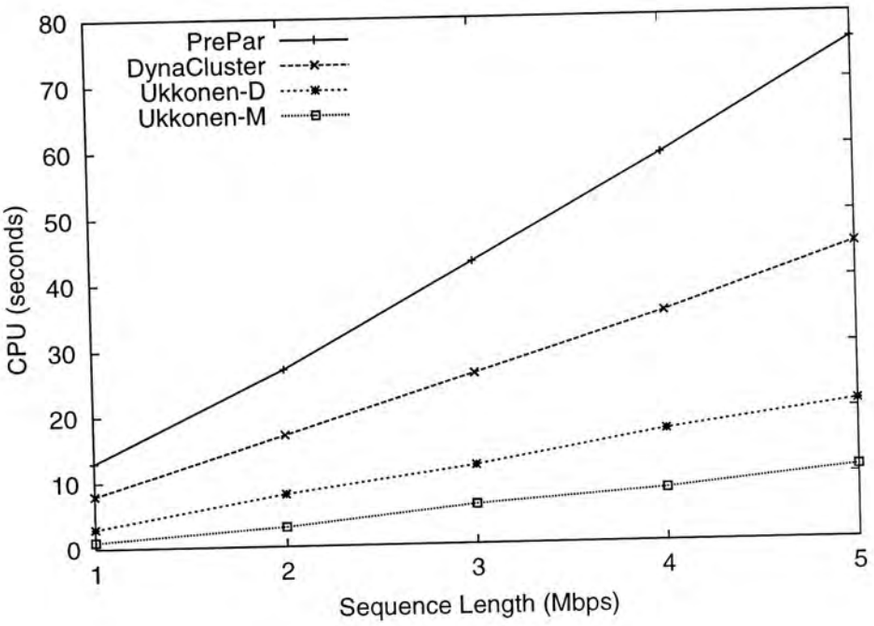
[1] www.cs.ucdavis.edu/~gusfield/strmat.html

whole DNA sequence in memory. It is in favor of PrePar because it needs to scan the whole sequence m-times. DynaCluster does not need to scan the whole sequence m times due to the assistance of prefix queues. Therefore, in the following, we do not include I/O time for scanning the whole sequence. The I/O cost for PrePar is the cost for accessing the suffix-tree buffer, and the I/O cost for DynaCluster is the cost for accessing both the suffix-tree buffer and the prefix-queue buffer.

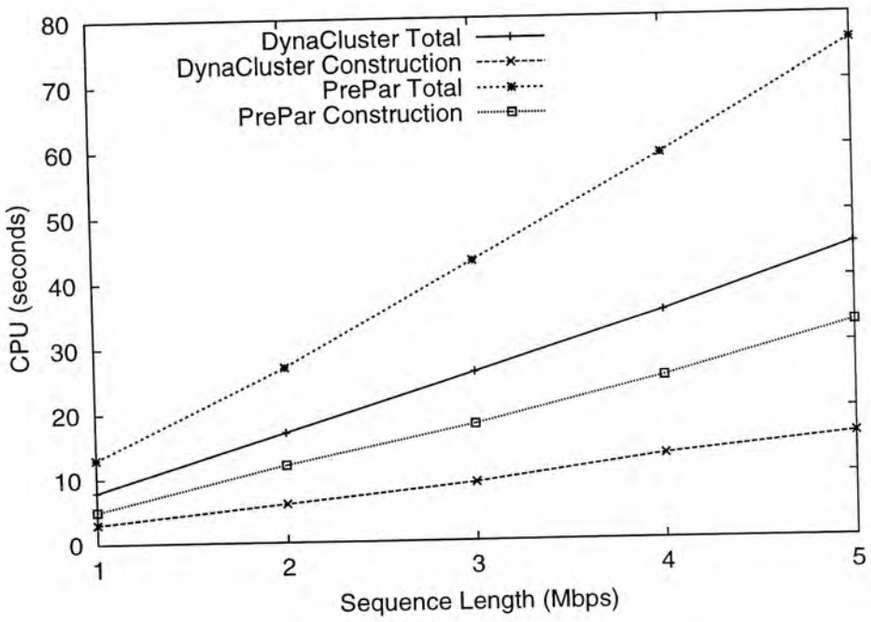
- Because the whole sequence is kept in main memory, for PrePar, we use the following settings. First the number of partitions, m, can be computed as $m = M_S/M_A = (d \times n)/M_A$. Note that we assume 60 bytes for an alphabet in a sequence S which is slightly larger than what we need to use in practice. For the same reason that the whole sequence is kept in main memory, we calculate the prefix length for PrePar as $l_{pp} = \log_{|\Sigma|} m$, based on the equation that $r = \lceil (|\Sigma|^l_{pp} - 1)/m \rceil = 1$. Recall $r = 1$ implies that PrePar will handle one prefix at a time. Fixed $r = 1$ and computing l, as shown in our extensive testing, can improve I/O costs. Therefore, it is in favor of releasing the burden of data skew. On the other hand, for DynaCluster, we fix the prefix length to be $l_{df} = 4$.
- When rebuilding suffix links, the testing setting for both DynaCluster and PrePar are the same.
- When answering queries using the disk-based suffix trees, the testing setting for both DynaCluster and PrePar are the same.

In the following testing, we show both CPU cost and I/O cost in seconds. The I/O cost in seconds is computed based on the total bytes we accessed from disk (8KB multiplied by the number of disk-page accesses) divided by 30MB per second transfer rate.[2]

[2] http://www.storage.ibm.com/hdd/ultra/36lzxdata.htm



(a) 4 Algorithms



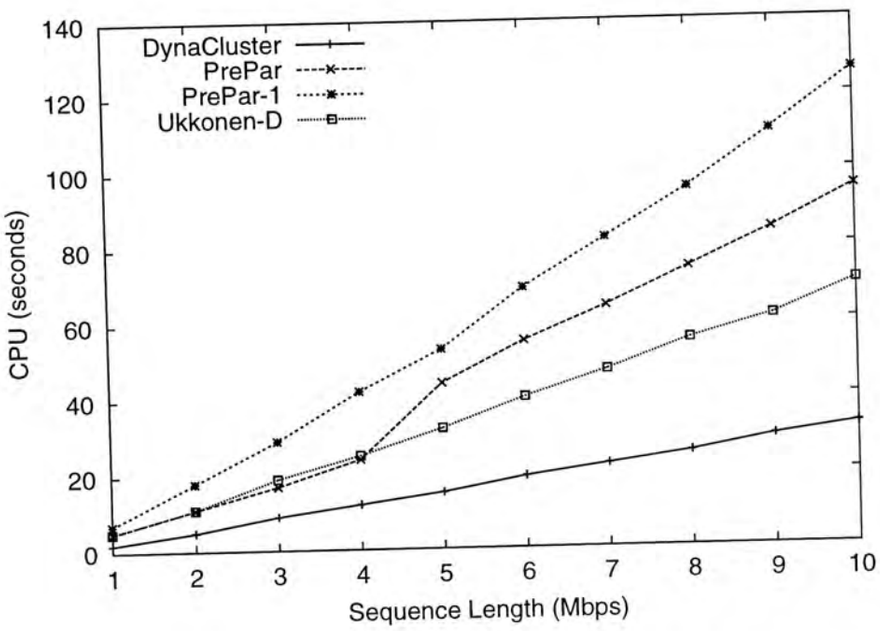
(b) `DynaCluster` vs `PrePar`

Figure 7.1: Suffix tree construction in memory

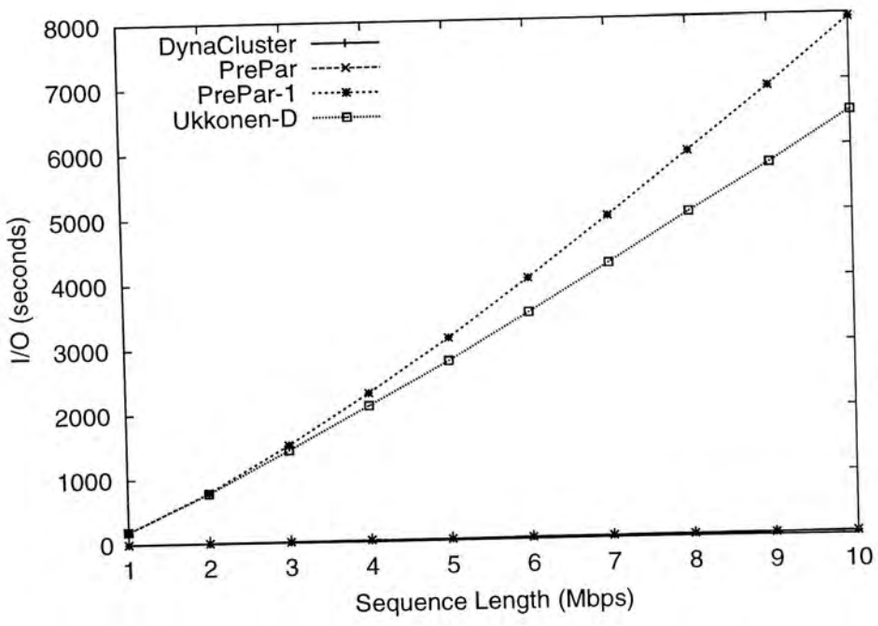
7.1 When Everything Can be Held In Memory

We first test the overhead of disk-based suffix-tree construction algorithms (with suffix-links rebuilt) with the in-memory based `Ukkonen-M` algorithm. We compare `DynaCluster`, `PrePar`, `Ukkonen-D` with `Ukkonen-M`, while varying the length of the `Chromosome-1` from 1Mbps to 5Mbps. Figure 7.1 shows the results of comparison. Note the costs for the three disk-based approaches: `Ukkonen-D`, `PrePar` and `DynaCluster` include both suffix-tree construction time and the suffix-link rebuilt time.

In Figure 7.1 (a), `Ukkonen-M` performs best, as expected. `Ukkonen-D` performs second best due to the overhead of handling disk-based pointers (indirectly accessing a node), and the overhead of buffer management (even though all the pages are in memory). The main difference between `PrePar` and `DynaCluster` is the cost of matching-and-edge-splitting. `Prepar` needs to follow the path from the root, and compare the suffix to be inserted with the edge-labels one-by-one, in order to find a mismatching position if any. In contract, `DynaCluster` does not need to do matching along the path from the root. It handles insertion in a cluster basis. In addition, it does not need to insert a prefix s_k^l into the cluster if the corresponding prefix queue is not empty, which means that the same prefix has already been inserted into the cluster. In this way, `DynaCluster` significantly reduces the overhead of searching the paths from the root. Therefore, `DynaCluster` outperforms `PrePar`, when all data can be held in memory. Figure 7.1 (b) further shows the costs for disk-based suffix-tree construction and suffix-link rebuilt. The costs for suffix-tree construction and suffix-link rebuilt are almost the same, when all data are in memory.



(a) CPU Cost



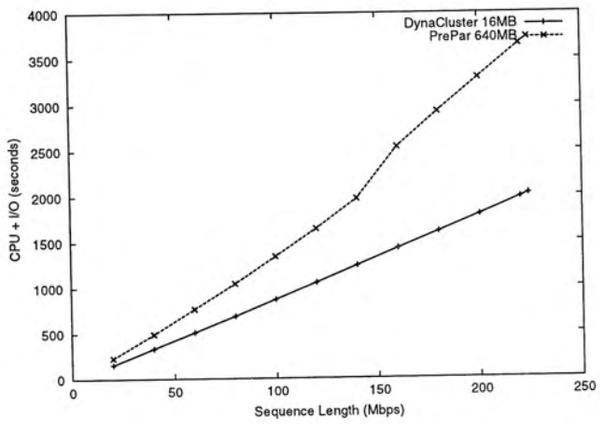
(b) I/O Cost

Figure 7.2: Disk-based suffix tree Construction for `Chromosome-X`

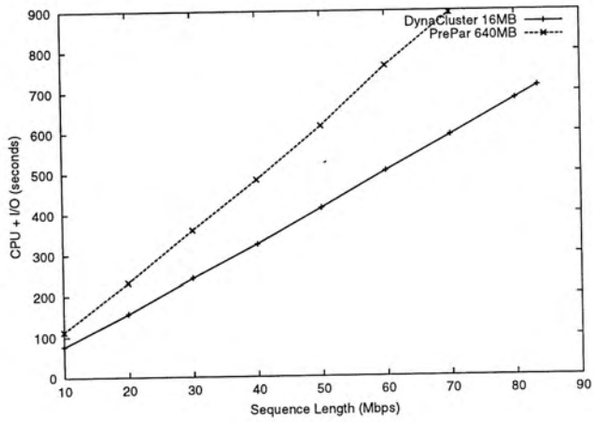
7.2 When Main Memory Is Limited

In the previous section, we studied 4 disk-based suffix tree construction algorithms, in comparison with the in-memory based algorithm, `Ukkonen-M`, when everything can be held in main memory. In this study, we focus on disk-based suffix tree construction only. Figure 7.2 illustrates the performance of the four disk-based algorithm, `DynaCluster`, `PrePar`, `Prepar-Suffix-1` and `Ukkonen-D`. We use 16MB main memory, and vary the length of `Chromosome-X` from 1Mbps to 10Mbps. Even though we only test a sequence up to 10Mbps, the 16MB main memory setting shows that we can obtain the similar trends when testing a long sequence with large main memory.

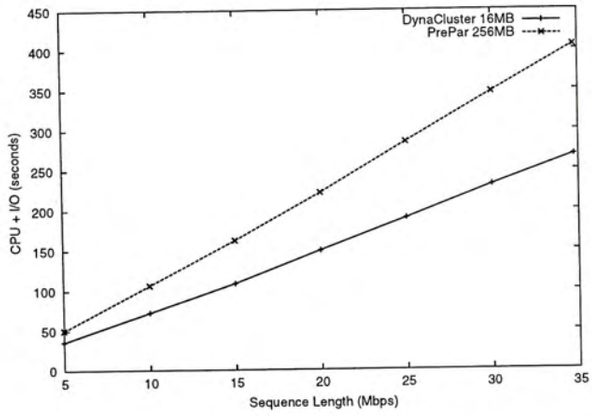
Figure 7.2 (a) and (b) show CPU cost and I/O cost, respectively. `DynaCluster` outperforms the others in terms of CPU cost and I/O cost. In terms of CPU cost, `Ukkonen-D`, `PrePar-1` and `DynaCluster` exhibit linearity. However, `PrePar` deviates from the trend at the point of 5Mbps, because of the change in the number of partitions. Recall the main memory size is fixed to be 16MB. Between 1 Mbps and 4Mbps, `PrePar` divides the sequence into 16 partitions. Starting from 5Mbps, the number of partitions for `PrePar` changes from 16 to 64. As a result, the number of sequence scans increases, and the cost for traversing the suffix-tree being built also increases. Although `Ukkonen-D` and `PrePar-1` are efficient in terms of CPU cost, their I/O cost is much worse than `PrePar` and `DynaCluster`. The I/O costs of `Ukkonen-D` and `PrePar-1` are over 6,000 seconds when the sequence length becomes 10Mbps, whereas the I/O costs for `PrePar` and `DynaCluster` are about 100 seconds. As a conclusion, `Ukkonen-D` and `PrePar-1` are not effective for dealing with large DNA sequences. Therefore, we do not report the results for `Ukkonen-D` and `PrePar-1` in the following testing, and will focus on `DynaCluster` and `PrePar`.



(a) Total Cost for `Chromosome-1`



(b) Total Cost for `Chromosome-18`



(c) Total Cost for `Chromosome-21`

Figure 7.3: `DynaCluster` (small memory) vs `PrePar` (large memory)

7.3 The Effectiveness of DNA Lengths with Fixed Memory Sizes

In this experimental study, we further examine the disk-based suffix tree construction cost for `DynaCluster` and `PrePar`, using three different DNA sequences: `Chromosome-1` (224Mbps), `Chromosome-18` (84Mbps), and `Chromosome-21` (35Mbps). For each of the three DNA sequences, we vary the length of DNA sequence incrementally with a fixed memory size, and in addition, we test the effectiveness of length of DNA sequences using three different memory sizes (small/middle/large). All the results are shown in Figure 7.5, Figure 7.6 and Figure 7.7, respectively. In Figure 7.4, we demonstrate the average suffix tree construction time using `chromosomes 1, 2, 10, 17, 18, and X`.

Before we discuss the details in the above three figures, in terms of total cost (CPU cost plus I/O cost), we show that `DynaCluster`, using 16MB memory size, outperforms `PrePar`, using 640MB memory size, for `Chromosome-1` (224Mbps) (Figure 7.3 (a)); `DynaCluster`, using 16MB memory size, outperforms `PrePar`, using 640MB memory size, for `Chromosome-18` (84Mbps) (Figure 7.3 (b)); and `DynaCluster`, using 16MB memory size, outperforms `PrePar`, using 256MB memory size, for `Chromosome-21` (35Mbps) (Figure 7.3 (c). Below, we summarize the results shown in Figure 7.5, Figure 7.6 and Figure 7.7.

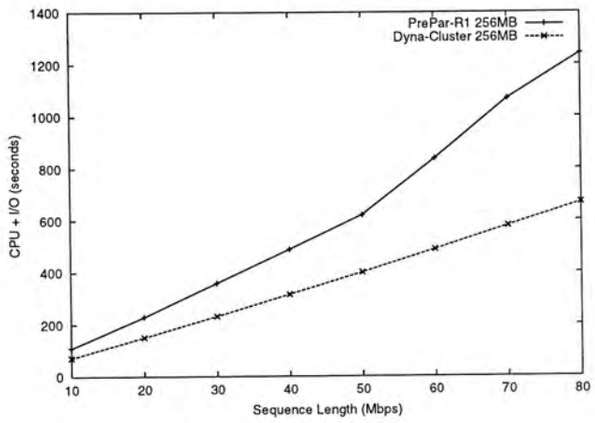
- In terms of the total cost (CPU cost plus I/O cost), `DynaCluster` significantly outperforms `PrePar`.
- In terms of CPU cost, both `PrePar` and `DynaCluster` share the similar behavior $O(n \log n)$, as indicated in [14]. However, `DynaCluster` significantly outperforms `PrePar` for the following reason. `PrePar` needs to traverse from the root one-letter-by-one when inserting a new suffix. The comparison cost for a huge sequence is not small. While the suffix tree becomes larger, more edge-splitting occurs, and therefore, when `PrePar` needs to traverse from one page to another, it might

also cause page-swapping by the underneath operating system. In contrast, `DynaCluster` does not need to traverse from the root, because it deals with clusters. `DynaCluster` does not need to insert a prefix s_k^l into the cluster if it finds that the corresponding prefix queue is not empty (which implies the same prefix s_k^l has already been inserted. The handling prefix queue might cause some overhead but is shown cost-effective.

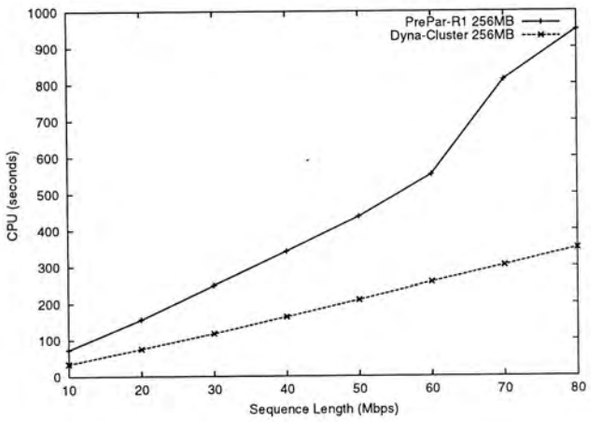
- In terms of I/O cost, `DynaCluster` shows a linear behavior, and is insensitive to the memory sizes, because it only needs memory for one-two clusters as its working space. `DynaCluster` uses two buffers. It uses a 8MB buffer for the suffix tree, and the rest for the prefix queues. For accessing prefix queues, it benefits from the sequential access. For both buffers, the pages are naturally swapped out from memory, and do not need to be swapped in again. The overhead of handling prefix queues does not occurs in `PrePar`. In occasions, `PrePar` outperforms `DynaCluster` slightly due to this reason, in terms of I/O cost. `PrePar` is influenced by two factors: the number of partitions and data skew. In Figure 7.5 (d), (e) and (f), Figure 7.6 (d), (e) and (f), and Figure 7.7) (d), (e) and (f), there are sharp drops of I/O cost for `PrePar`, which is caused by the increase of partition numbers, for example, from 4 to 16, or from 16 to 64. At the switching point of partition numbers, `PrePar` performs well. However, while the length of DNA sequence increases, data skew becomes sever. The I/O cost increases exponentially.

7.4 The Effectiveness of Memory Sizes

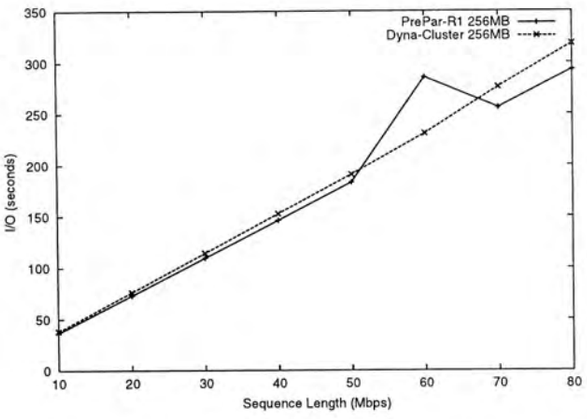
In the previous section, we fixed memory size, and varied the length of DNA sequences. In this section, we fix the length of DNA sequences, but vary the memory size. We construct a suffix tree for the entire sequence of `Chromosome-10` and `Chromosome-21`, respectively. The results are shown in Figure 7.8 and Figure 7.9. In terms of total cost (CPU plus I/O), `DynaCluster`



(a) Total Cost ($M_A = 256$)

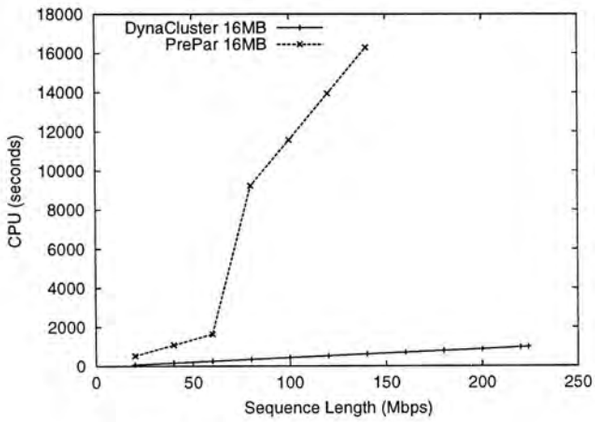


(b) CPU Cost ($M_A = 256$)

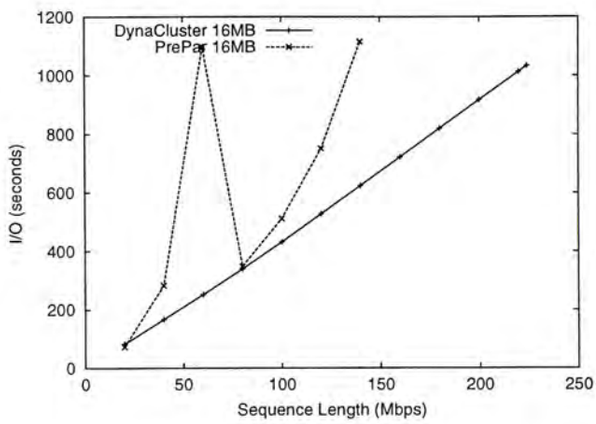


(c) I/O Cost ($M_A = 256$)

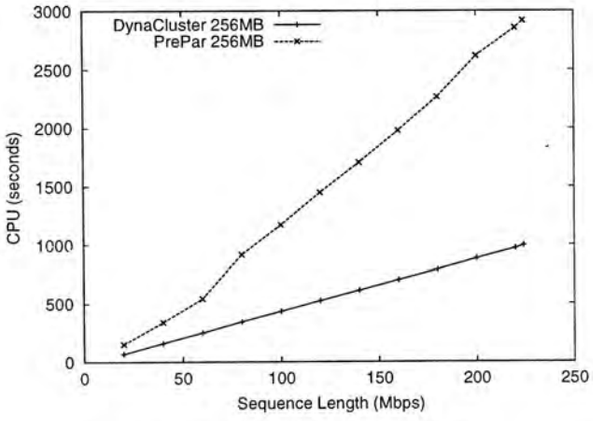
Figure 7.4: Average cost for constructing suffix trees for chromosomes 1, 2, 10, 17, 18, and X



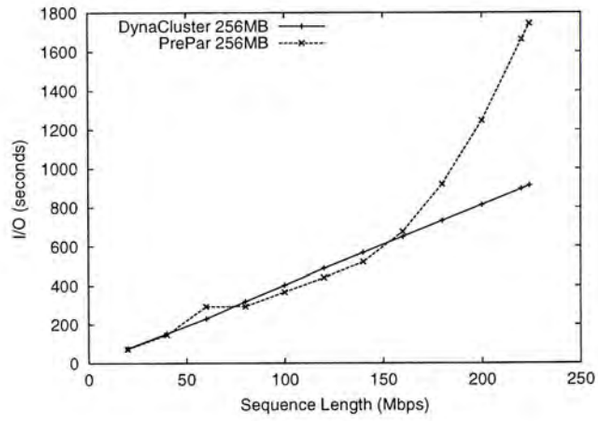
(a) CPU Cost ($M_A = 16$)



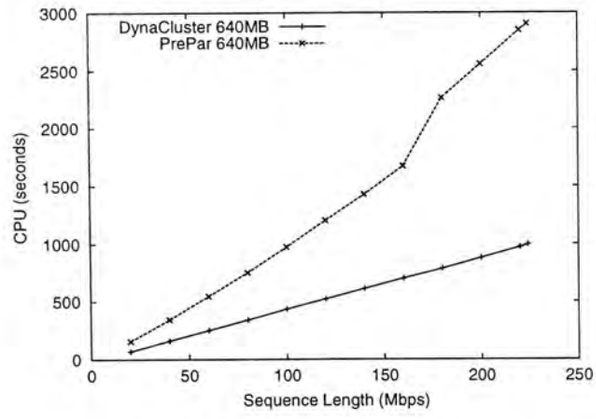
(b) I/O Cost ($M_A = 16$)



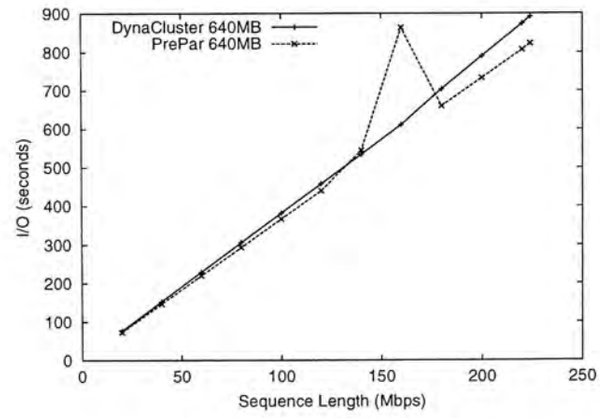
(c) CPU Cost ($M_A = 256$)



(d) I/O Cost ($M_A = 256$)



(e) CPU Cost ($M_A = 640$)



(f) I/O Cost ($M_A = 640$)

Figure 7.5: Disk-based suffix tree construction for `Chromosome-1` (224Mbps)

significantly outperforms PrePar. DynaCluster shows the linear behavior while the memory size increases. It shows that it only needs minimum 16MB memory space to construct a disk-based suffix tree. On the other hand, PrePar shows its zigzag behavior. On the point the partition number changes, the I/O cost for PrePar significantly reduces. However, the effectiveness of partitioning becomes small, when the data skew becomes sever gradually. Note, increasing partition number does not improve CPU cost.

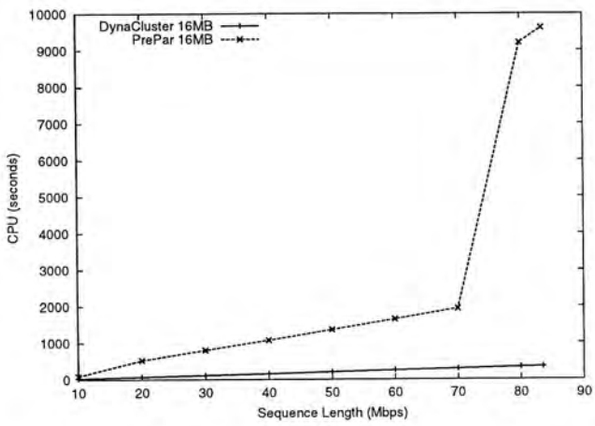
7.5 Answering q-Length Exact Sequence Matching Queries

We also conduct testing to test the quality of the disk-based suffix trees constructed by PrePar and DynaCluster. Figure 7.10 shows the results from batches of 10,000 random exact sequence matching queries of different lengths on Chromosome-18. In Figure 7.10 (a), it shows the number of results from queries of length q, from $q = 8$ to $q = 15$. We also conduct testing for $q > 15$ exact sequence matching queries, but the number of results is too small, because we randomly generated queries. The I/O cost of each exact sequence matching query consists of two parts. The first is the cost to walk down the suffix tree until a match or mismatch is found, and the other is the cost of traversing down the tree to gather all the suffix positions s_k that satisfy the q-length exact sequence matching query. DynaCluster significantly outperforms PrePar. When no results are found, DynaCluster outperforms PrePar by two times. We also test the exact sequence matching queries using different disk-based suffix trees constructed using different memory sizes. The quality of the disk-based suffix trees constructed by DynaCluster outperforms the disk-based suffix trees constructed by Prepar. In particular, the disk-based suffix tree constructed by DynaCluster, using 16MB memory, outperforms all the disk-based suffix trees constructed by PrePar using 16MB, 256MB and 640MB memory. For PrePar, it is important to know that it does not necessarily guarantee a better performance for answering

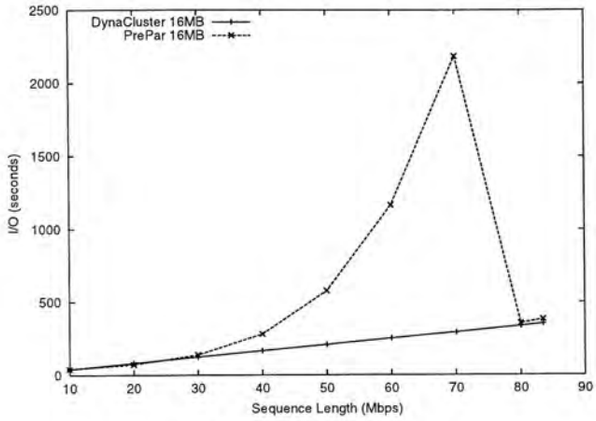
exact sequence matching queries, when it uses more memory space to construct a disk-based suffix tree. Figure 7.10 (b), (c) and (d) show the query performance for three different sets of disk-based suffix trees (using 16MB, 256MB and 640MB). The query performance for the disk-based suffix tree constructed using 640MB for `PrePar` is worse than that of the disk-based suffix trees constructed using 16MB and 256MB. It is because that when `PrePar` uses larger memory to construct a disk-based suffix tree, the number of partitions becomes smaller and therefore the size of each partition becomes larger. Consequently, the random access caused by edge-splitting for a larger partition becomes much severer. We conclude that our suffix tree is effective and efficient in answering exact sequence matching queries.

7.6 Suffix Link Rebuilt

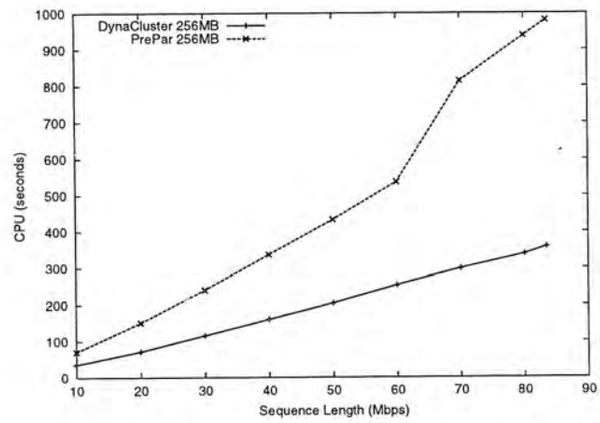
The main cost for suffix link rebuilt is the cost incurs for the preparation phase for the least common ancestors (lca) queries. The preparation requests to traverse the constructed disk-based suffix tree twice. Figure 7.11 and Figure 7.12 show the quality of the disk-based suffix trees constructed by `DynaCluster` is much better than that of the disk-based suffix trees constructed by `PrePar`. The suffix-link rebuilt shows the same behavior when the disk-based suffix-tree is constructed. `DynaCluster` outperforms `PrePar` in all memory configurations. It is important to know that a disk-based suffix tree with suffix links can support approximate query matchings [11].



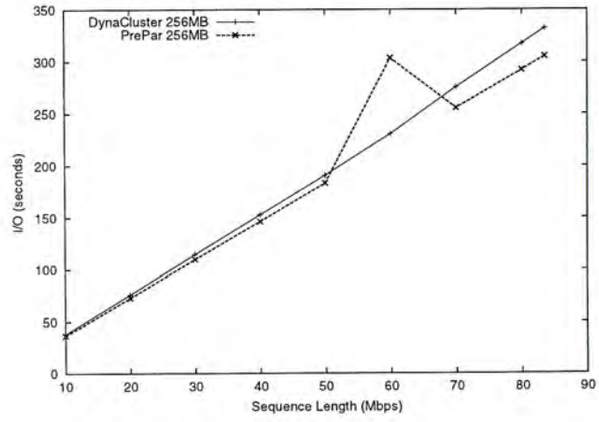
(a) CPU Cost ($M_A = 16$)



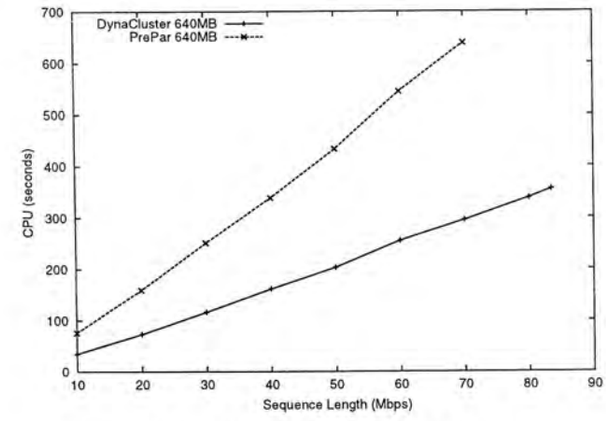
(b) I/O Cost ($M_A = 16$)



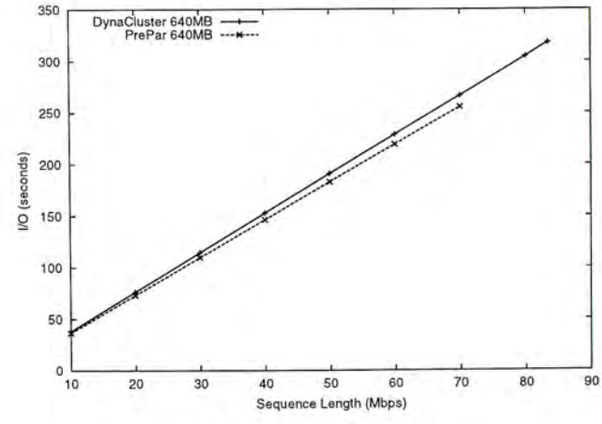
(c) CPU Cost ($M_A = 256$)



(d) I/O Cost ($M_A = 256$)

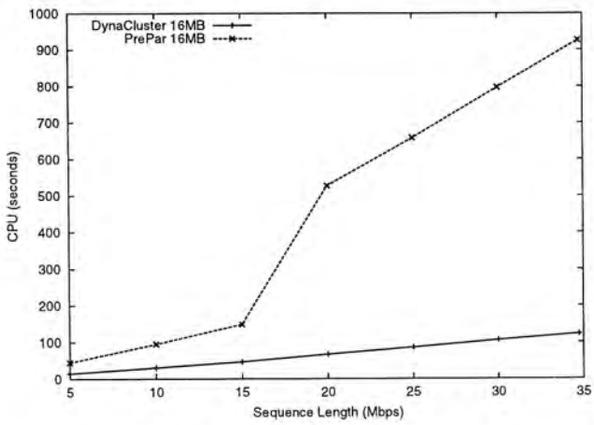


(e) CPU Cost ($M_A = 640$)

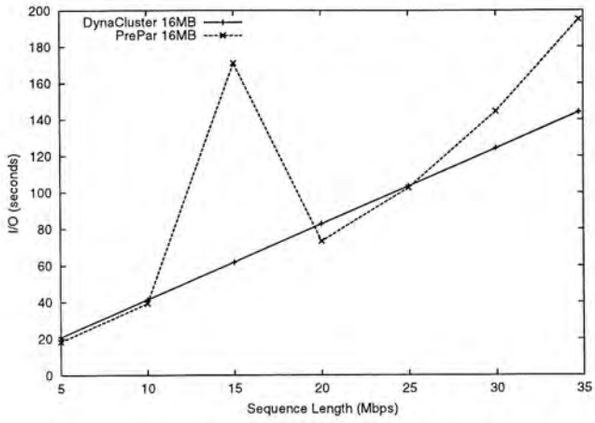


(f) I/O Cost ($M_A = 640$)

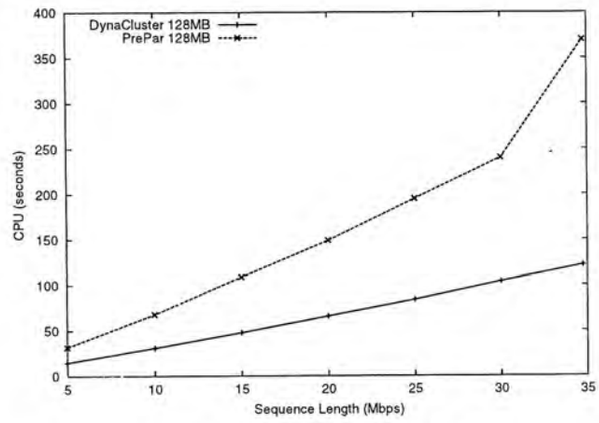
Figure 7.6: Disk-based suffix tree construction for `Chromosome-18` (84Mbps)



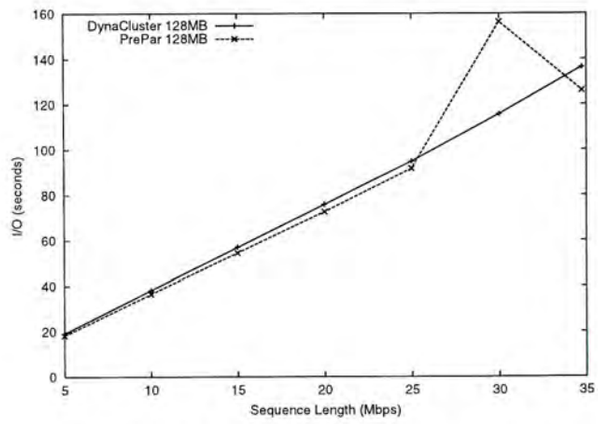
(a) CPU Cost ($M_A = 16$)



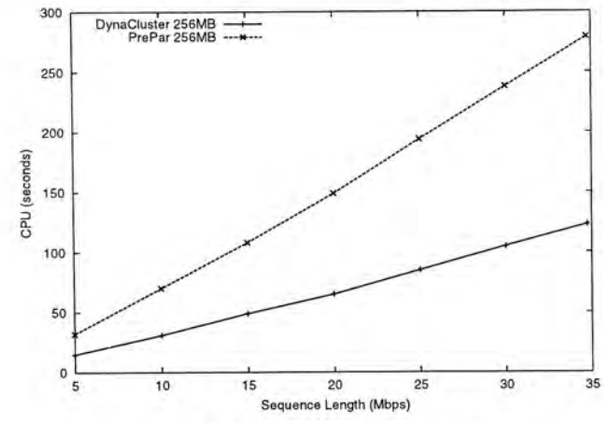
(b) I/O Cost ($M_A = 16$)



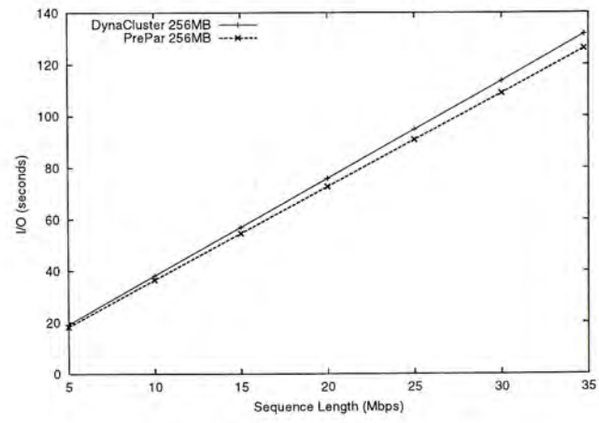
(c) CPU Cost ($M_A = 128$)



(d) I/O Cost ($M_A = 128$)

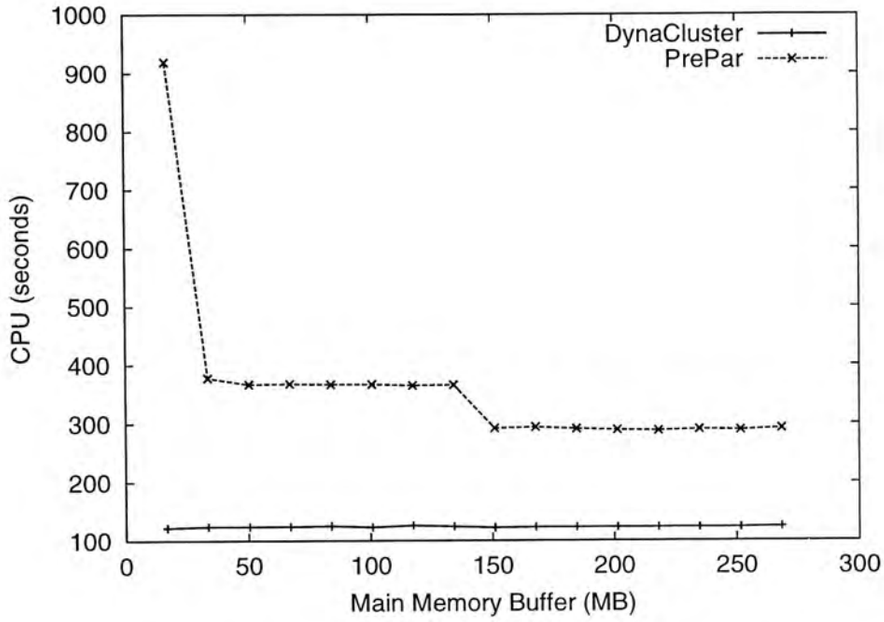


(e) CPU Cost ($M_A = 256$)

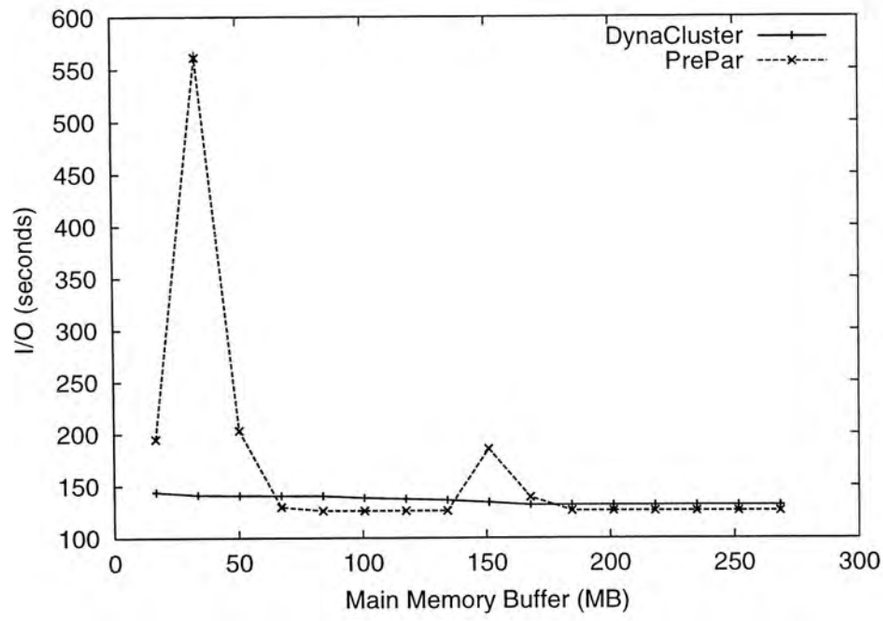


(f) I/O Cost ($M_A = 256$)

Figure 7.7: Disk-based suffix tree construction for `Chromosome-21` (35Mbps)

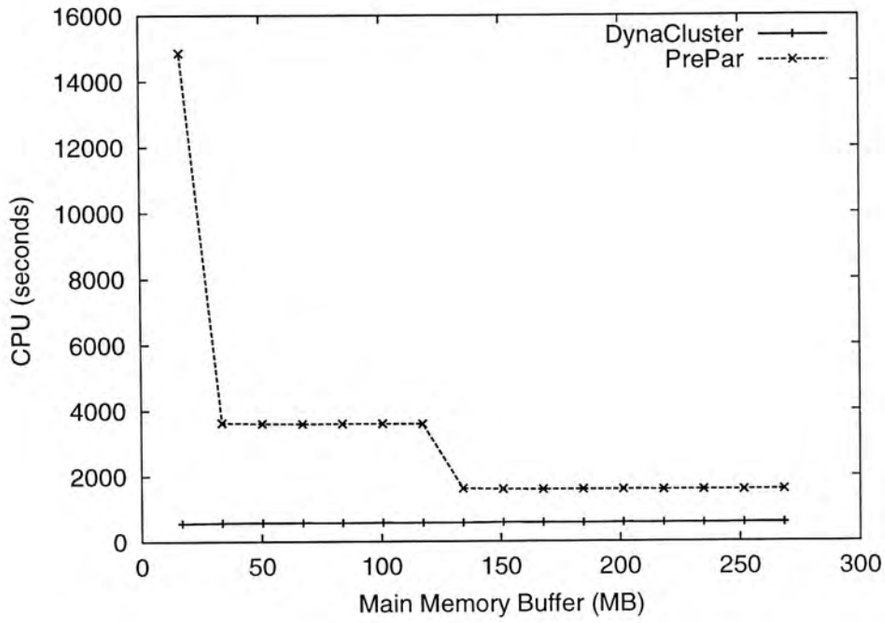


(a) CPU Cost

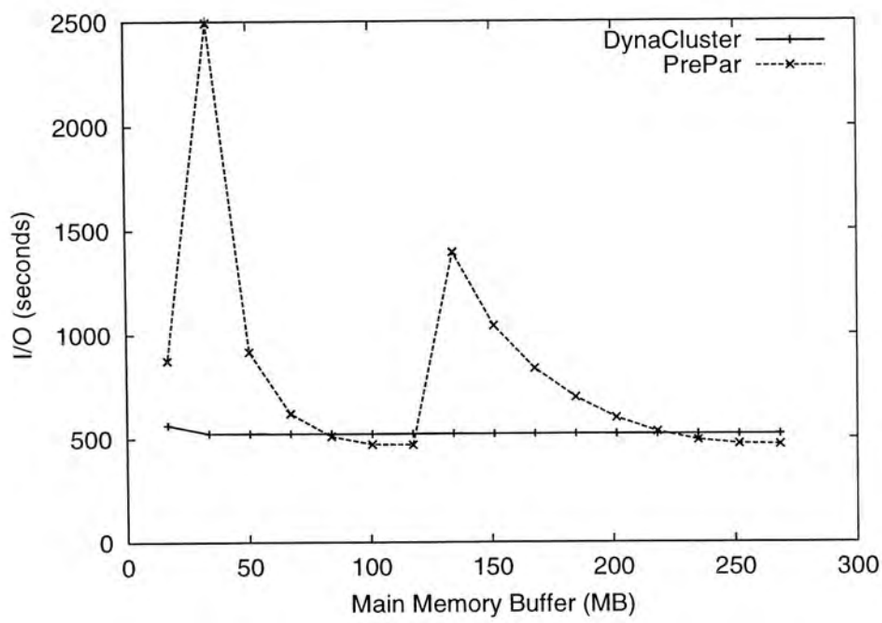


(b) I/O Cost

Figure 7.8: Disk-based Suffix tree construction for `Chromosome-21` (35Mbps) using different memory Sizes

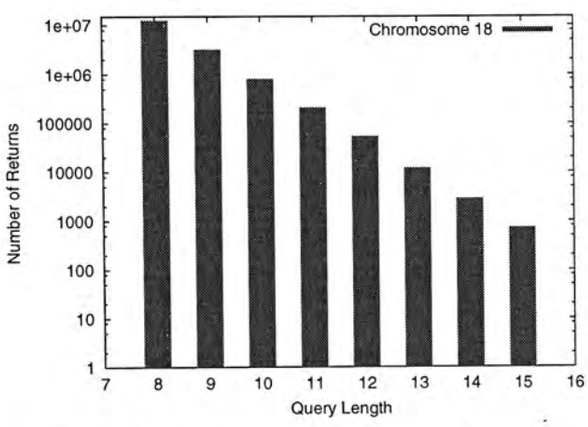


(a) CPU Cost

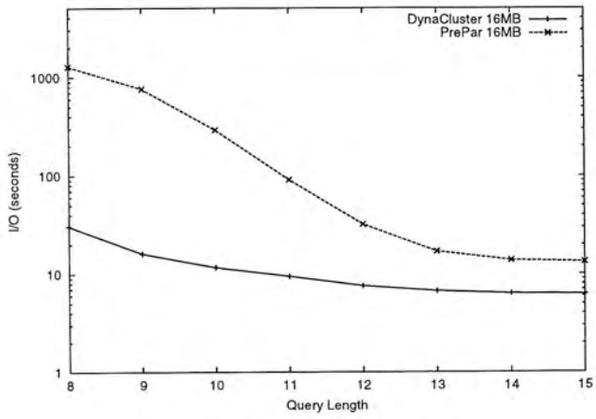


(b) I/O Cost

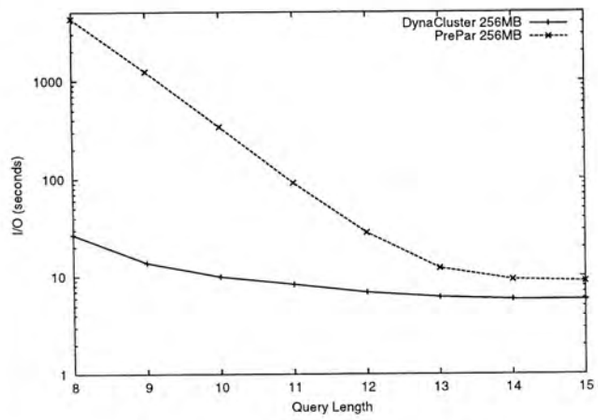
Figure 7.9: Disk-based suffix tree construction for `Chromosome-10` (128Mbps) using different memory sizes



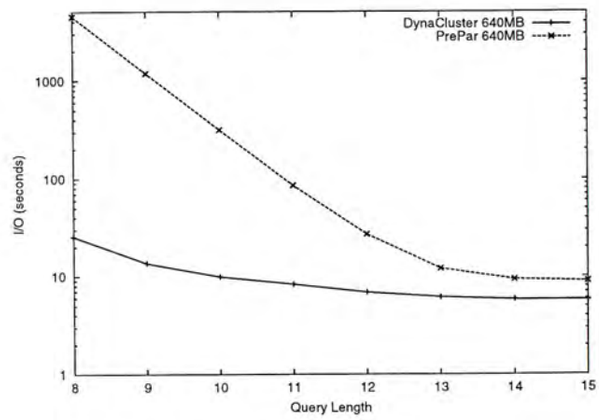
(a) Number of returns for q-length queries



(b) CPU Cost ($M_A = 16$)

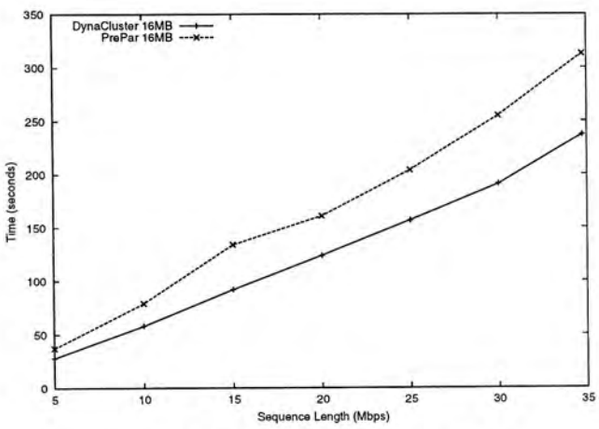


(c) CPU Cost ($M_A = 256$)

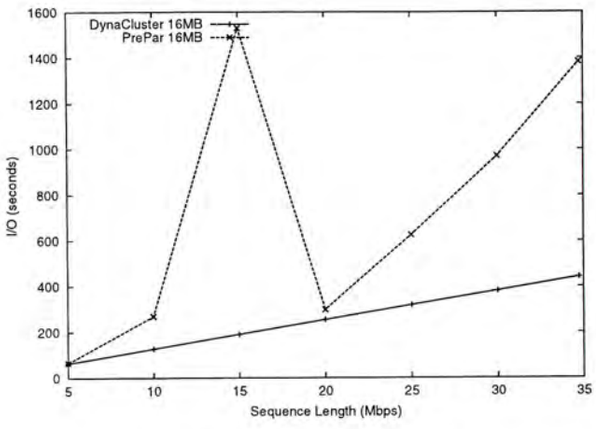


(d) CPU Cost ($M_A = 640$)

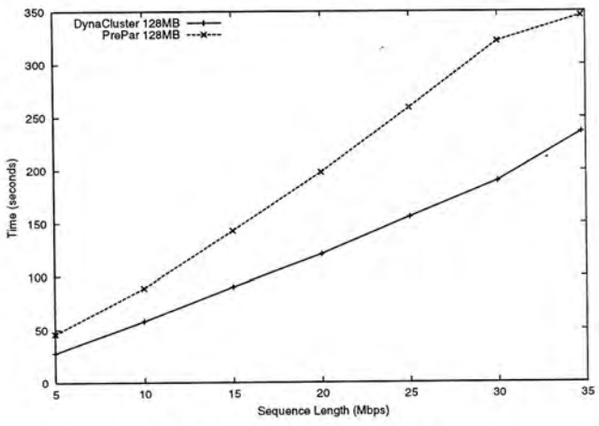
Figure 7.10: Batches of 10,000 queries against `Chromosome-18` (85Mbps)



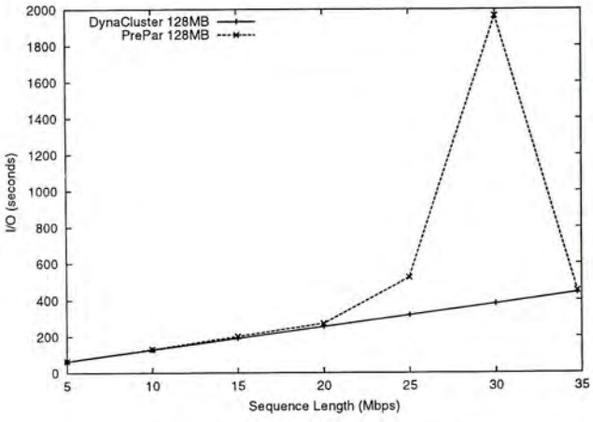
(a) CPU Cost ($M_A = 16$)



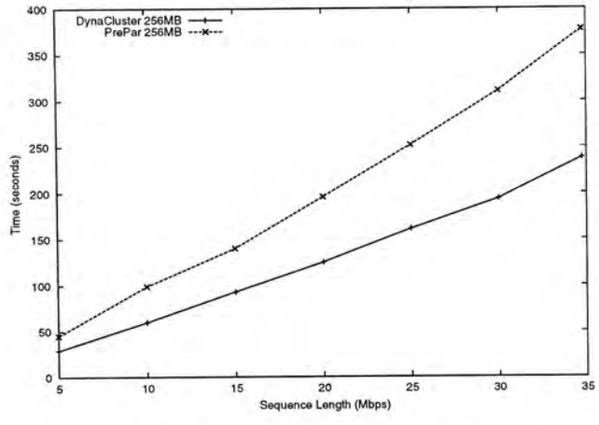
(b) I/O Cost ($M_A = 16$)



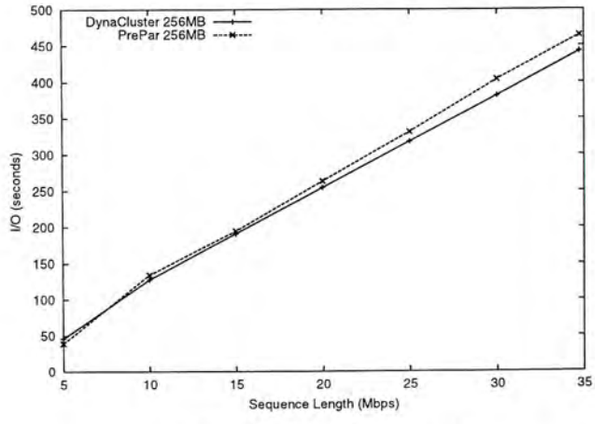
(c) CPU Cost ($M_A = 128$)



(d) I/O Cost ($M_A = 128$)

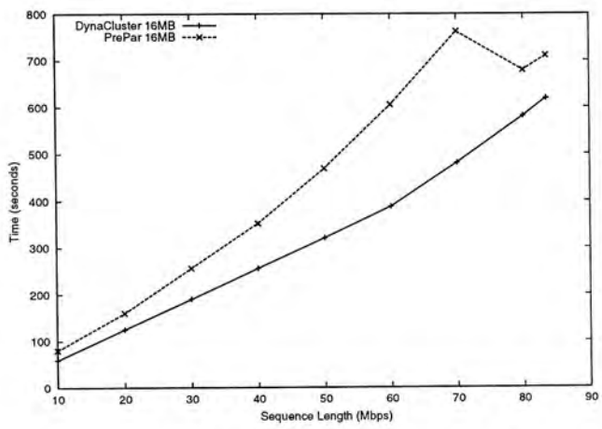


(e) CPU Cost ($M_A = 256$)

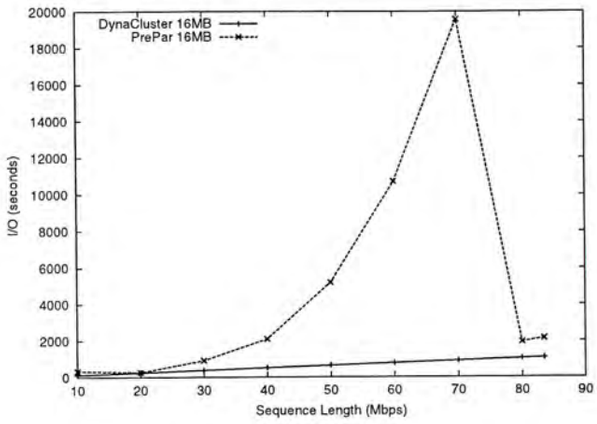


(f) I/O Cost ($M_A = 256$)

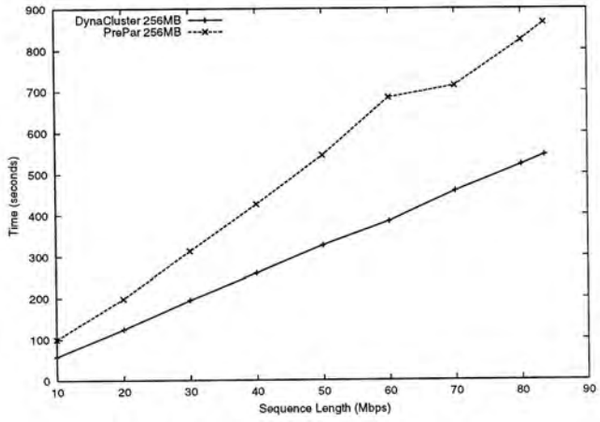
Figure 7.11: Suffix link rebuilt for `Chromosome-21` (35Mbps)



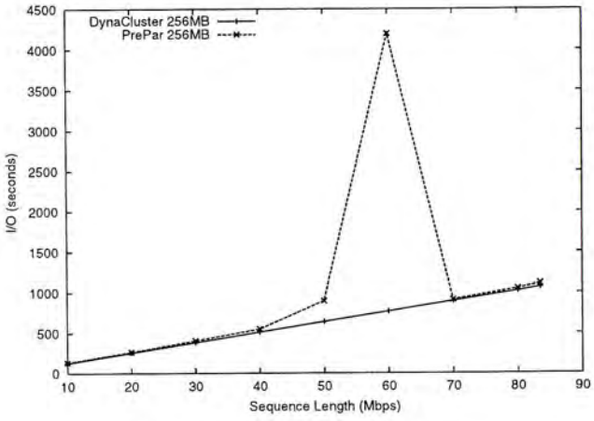
(a) CPU Cost ($M_A = 16$)



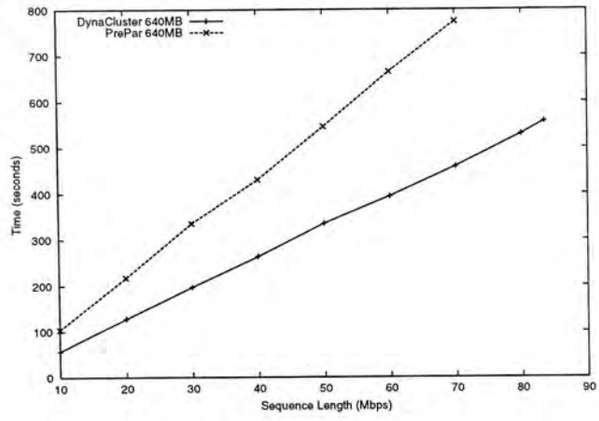
(b) I/O Cost ($M_A = 16$)



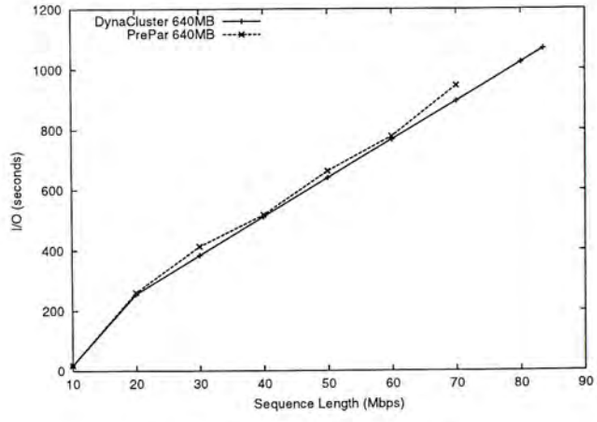
(c) CPU Cost ($M_A = 256$)



(d) I/O Cost ($M_A = 256$)



(e) CPU Cost ($M_A = 640$)



(f) I/O Cost ($M_A = 640$)

Figure 7.12: Suffix link rebuilt for `Chromosome-18` (84Mbps)

Chapter 8

Conclusions and Future Works

8.1 Conclusions

In this thesis, we discuss the disk-based suffix-tree construction algorithms. Suffix-trees can be used to perform q-length exact sequence matching and suffix-links can be rebuilt using constant time least common ancestors. We point out that the `PrePar-Suffix`, which uses pre-partitioning techniques, suffers from the problems of edge-splitting, random access, and data skew. We address the problems with our novel disk-based suffix tree construction algorithm, `DynaCluster-Suffix`. The `DynaCluster-Suffix` algorithm uses a very small working memory to construct large disk-based suffix trees for large DNA sequences. In addition to our stringent memory requirement, the clustering property of the disk-based suffix trees being constructed with `DynaCluster-Suffix` is much better than those being constructed with `PrePar-Suffix`. It explains why exact string matching and suffix links rebuild on `DynaCluster-Suffix` are more efficient. To conclude, the disk-based suffix trees `DynaCluster-Suffix` constructed using 16 MB memory outperforms the disk-based suffix trees `PrePar-Suffix` constructed using 640MB memory, in terms of both suffix tree construction cost and query processing cost.

8.2 Future Works

There are a number of possible research directions in extending this research. Firstly, we can explore more suffix tree applications using `DynaCluster-Suffix`. Currently, we implemented the q-length exact sequence matching. It is possible to extend to approximate sequence matching. The k-difference approximate matching, which allows insertion, deletion, and substitution, can be implemented using least common ancestors and suffix trees. Moreover, repetitive structures like tandem repeats, palindromes can be found using suffix trees, too. Secondly, we can extend our implementation to protein sequences, which contains 20 alphabets. Finally, we can create an online database index for genomes such that biologists can make use of our research to investigate the human biology and diseases.

Bibliography

[1] R. Baeza-Yates and G. Navarro. A new indexing method for approximate string matching. In *CPM99, LNCS 1645*, 1999.

[2] R. Baeza-Yates and G. Navarro. A hybrid indexing method for approximate string matching. In *Journal of Discrete Algorithms*, 2000.

[3] P. Bieganski. *Genetic Sequence Data Retrieval and Manipulation based on Generalized Suffix Trees.* PhD thesis, University of Minnesota, USA, 1995.

[4] Robert S. Boyer and J. Strother Moore. A fast string searching algorithm. *Communcation of ACM*, pages 762–772, 1977.

[5] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* The MIT Press, 1989.

[6] Martin Farach. Optimal suffix tree construction with large alphabets. In *Proceedings of the 38th Annual Symposium on the Foundation of Computer Science*, 1997.

[7] Martin Farach, Paolo Ferragina, and S. Muthukrishnan. Overcoming the memory bottleneck in suffix tree construction. In *39th Symposium on Foundations of Computer Science*, 1998.

[8] Martin Farach and S. Muthukrishnan. Optimal logarithmic time randomized suffix tree construction. In *Proc. of 23rd International Coloquium on Automata Languages and Programming*, pages 550–561, 1996.

[9] R. Giegerich and S. Kurtz. From ukkonen to mccreight and weiner: A unifying view of linear-time suffix tree construction. In *Algorithmica*, pages 331–353, 1997.

[10] Robert Giegerich, Stefan Kurtz, and Jens Stoye. Efficient implementation of lazy suffix trees. In *In Proceedings of the Third Workshop on Algorithmic Engineering (WAE99)*, 1999.

[11] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences Computer Science and Computational Biology.* Cambridge University Press, 1997.

[12] Dan Gusfield and Jens Stoye. Linear time algorithms for finding and representing all tandem repeats in a string. Technical report, Computer Science Department, University of California, Davis, 1998.

[13] Dov Harel and Robert Endre Tarjan. Fast algorithms for finding nearest common ancestors. In *SIAM Journal of Computing*, volume 13, pages 338–355, May 1984.

[14] Ela Hunt, Malcolm P. Atkinson, and Robert W. Irving. Database indexing for large dna and protein sequence collections. In *VLDB Journal*, 2001.

[15] Ela Hunt, Robert W. Irving, and Malcolm Atkinson. Persistent suffix trees and suffix binary search trees as dna sequence. Technical report, Department of Computing Science, University of Glasgow, October 4, 2000.

[16] J. Karkkainen and Esko Ukkonen. Sparse suffix trees. In *In Proceedings of COCOON*, 1996.

[17] Juha Kärkkäinen. Suffix cactus: A cross between suffix tree and suffix array. In *The 6th Symposium on Combinatorial Pattern Matching*, 1995.

[18] D. E. Knuth, J. H. Morris, and V. R. Pratt. Fast pattern matching in strings. *SIAM. Journal of Computing*, 6, 1977.

[19] Stefan Kurtz. Reducing the space requirement of suffix trees. *Software Practice and Experience*, 29(13):1149–1171, 1999.

[20] Gonzalo Navarro. A guided tour to approximate string matching. *ACM Computing Surveys*, 33(1):31–88, 2001.

[21] Baruch Schieber and Uzi Vishkin. On finding lowest common ancestors: Simplication and parallelization. In *SIAM Journal of Computing*, volume 17, pages 1253–1262, December 1988.

[22] W. Szpankowski. Asymptotic properties of data compression and suffix trees. In *IEEE Transactions on Information Theory*, 1993.

[23] P. Weiner. Linear pattern matching algorithm. In *Proceeding, 14 IEEE Symposium on Switching and Automata Theory*, 1973.

[24] Aaron D. Wyner and Jacob Ziv. Some asymptotic properties of the entropy of a stationary ergodic data source with applications to data compression. In *IEEE Transactions on Information Theory. Vol. 35, No. 6*, 1989.